

Controlling Interactions with Libraries in Android Apps Through Runtime Enforcement

OLIVIERO RIGANELLI*, University of Milano - Bicocca, Italy

DANIELA MICUCCI, University of Milano - Bicocca, Italy

LEONARDO MARIANI, University of Milano - Bicocca, Italy

Android applications are executed on smartphones equipped with a variety of resources that must be properly accessed and controlled, otherwise the correctness of the executions and the stability of the entire environment might be negatively affected. For example, apps must properly acquire, use, and release microphones, cameras, and other multimedia devices otherwise the behaviour of the apps that use the same resources might be compromised.

Unfortunately, several apps do not use resources correctly, for instance due to faults and inaccurate design decisions. By interacting with these apps users may experience unexpected behaviours, which in turn may cause instability and sporadic failures, especially when resources are accessed.

In this paper, we present an approach that lets users *protect* their environment from the apps that use resources improperly by enforcing the correct usage protocol. This is achieved by using *software enforcers* that can observe executions and change them when necessary. For instance, enforcers can detect that a resource has been acquired but not released, and automatically perform the release operation, thus giving the possibility to use that same resource to the other apps.

The main idea is that software libraries, in particular the ones controlling access to resources, can be augmented with enforcers that can be activated and deactivated on demand by users to protect their environment from unwanted app behaviours. We call the software libraries augmented with one or more enforcers *proactive libraries* because the activation of the enforcer decorates the library with *proactive behaviours* that can guarantee the correctness of the execution despite the invocation of the operations implemented by the library. For example, enforcers can detect that a resource has not been released on time and proactively release it.

Our experimental results with 27 possible misuses of resources in real Android apps reveal that proactive libraries are able to effectively correct library misuses with negligible runtime overheads.

CCS Concepts: • **Computer systems organization** → **Self-organizing autonomic computing**; • **Software and its engineering** → **Software fault tolerance**; *Software libraries and repositories*;

Additional Key Words and Phrases: proactive library, self-healing, Android, resource leak, resource usage, policy enforcement, runtime enforcement

ACM Reference Format:

Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. 2019. Controlling Interactions with Libraries in Android Apps Through Runtime Enforcement. 1, 1 (November 2019), 29 pages. <https://doi.org/0000001.0000001>

*This is the corresponding author

Authors' addresses: Oliviero Riganelli, University of Milano - Bicocca, Viale Sarca 336, Milan, MI, 20126, Italy, oliviero.riganelli@unimib.it; Daniela Micucci, University of Milano - Bicocca, Viale Sarca 336, Milan, MI, 20126, Italy, daniela.micucci@unimib.it; Leonardo Mariani, University of Milano - Bicocca, Viale Sarca 336, Milan, MI, 20126, Italy, leonardo.mariani@unimib.it.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.

© 2019 Association for Computing Machinery.

XXXX-XXXX/2019/11-ART \$15.00

<https://doi.org/0000001.0000001>

1 INTRODUCTION

Users of mobile devices can easily and quickly access to a huge number of software applications through well-established digital marketplaces, such as the Android's Google Play¹, the Apple's App Store², and the Microsoft's Windows Phone App Store³. These marketplaces are freely populated with applications of various quality, ranging from professional apps to apps implemented by hobbyists. The openness of the marketplaces facilitates software distribution but also raises relevant reliability issues. In fact many of these apps are often not thoroughly validated and may cause troubles once installed and executed in a device.

To mitigate these problems, the marketplaces might be equipped with mechanisms to limit the proliferation of unreliable and insecure apps. For example, users can rate and review apps, so that unsafe and unreliable apps become less and less popular and fewer people try to install and use them [27]. Ecosystems may also implement static analysis routines to reject largely inadequate apps [29].

Although these mechanisms can be helpful to prevent the indiscriminate proliferation of undesired apps, customers regularly report issues with the apps downloaded from the marketplaces [6, 7, 51, 54, 55]. A relevant portion of the problems experienced by the users is related to the way apps interact with the resources available in a device (e.g., the camera, the microphone, and the wifi antenna). For instance, many apps fail to properly acquire and release resources, causing efficiency and energy problems [6, 7, 55]. In some cases, inaccurate interactions with the resources may even cause problems across apps. For instance, an app that does not release the camera every time its execution is suspended may prevent the other apps from acquiring the camera [6, 55].

Since many of these annoying problems can be addressed by forcing the faulty apps to satisfy the expected resource usage protocol, we studied how to design *software enforcers* that can monitor executions and change the behaviour of the apps when necessary. To make the solution broadly applicable, software enforcers do not require access to the source code of the app, but work in a black-box fashion. Moreover, they could be speculatively activated or deactivated by users. For instance, enforcers might be activated after some applications presented some misbehaviours.

Interestingly, the incorrect interaction between an app and a resource can be often recognized by looking at both the usage of the library that controls the access to the resource and the lifecycle of the components of the app that interact with the library, for instance the lifecycle of an activity⁴. For example, if the user moves to background an app that is using the microphone to record some audio, the app must immediately release the microphone, otherwise the other apps might be unable to interact with the microphone since it would be occupied by the inactive app, and the device may unnecessarily keep the microphone active consuming extra battery. At the level of the interaction between the app and the library this means that once an app has invoked the method `startRecording()` on the Android library class `AudioRecord`, it must release the microphone by invoking the method `release()` every time a call to the `onStop()` callback method is generated by the Android framework (`onStop()` is a callback method that apps implement to define the operations that must be performed before an activity becomes invisible to the user). Unfortunately, it is not always the case that apps are implemented releasing and acquiring resources coherently with the lifecycle of the activities [6, 34, 55].

Enforcers can intercept interactions between apps and libraries to automatically fix these executions. Since enforcers are aware of the usage policies relevant to Android libraries, they are logically associated with libraries, that is, each library is decorated with a number of enforcers each one enforcing a different *policy*. The presence of the enforcers turns regular libraries into *proactive libraries*, which are libraries that can perform operations even when the methods of the libraries are not invoked (e.g., because the enforcer has to proactively execute operations

¹<https://play.google.com/store>

²<https://www.apple.com>

³<https://www.microsoft.com/en-in/store/apps/>

⁴Android apps are composed of multiple components called activities: <https://developer.android.com/guide/components/activities>

to fix a detected problem). For example, when the `onStop()` callback is generated, the enforcer associated with the library for audio recording is triggered to check if the microphone has been released. If not, the enforcer may force the release of the microphone, and automatically reassign it to the activity once the activity becomes visible to the user again.

Since a regular library does not need to be modified to be turned into a proactive library, enforcers can be easily added to existing libraries, without the need of designing libraries as proactive libraries from the very beginning. This eases the incremental adoption of the technology. Moreover, any party may contribute to the definition of the enforcers for a given library and not only the developers of the library. This aspect may facilitate the proliferation of the enforcers. App developers can keep using the usual testing and analysis methods to fix their apps. The design and implementation of enforcers is for library developers and in general for developers who want to contribute to the Android ecosystem, while the runtime activation of the enforcers is under the control of the final users.

In addition to studying how to turn regular libraries into proactive libraries that can be exploited to make software systems more reliable, we defined a model-based approach for both the definition of the enforcers and the automatic generation of their implementation, that we call *proactive module*, for the Android environment. Although the concepts of enforcers and proactive libraries are general, we studied them in the context of the Android ecosystem because many non-trivial and rapidly evolving APIs for resource management are available and problems with resources are frequent [6, 7, 55].

This work extends our previous work on proactive libraries [44] by (i) introducing a process, a modelling environment, and a code generation strategy to automatically generate the proactive libraries from a specification of the enforcers, (ii) extending the modelling language used to specify the enforcers with special operations for restoring resources and with the possibility to distinguish if an action taken by an enforcer must be executed before or after an operation performed by the monitored software, (iii) increasing the sophistication of the proactive module to consider the status of Android components, enforcement strategies, and resources simultaneously, (iv) providing a more detailed presentation of the approach and additional background material, (v) extending the empirical evaluation with 15 additional real-world cases from popular Android markets and open-source Android community; and (vi) studying the scalability of the proactive modules and the usefulness of the model-based approach compared to the manual implementation of the enforcers.

The paper is organized as follows. Section 2 provides background information about edit automata, which is the formalism used in this paper to specify enforcers, model-driven software development, which is the methodology used to specify and generate the proactive libraries, and Xposed, which is the framework we used to implement the proactive modules for Android. Section 3 introduces a running example that is used throughout the paper to exemplify the approach. Section 4 presents the concept of proactive library and the process for implementing it. Section 5 describes how to formally define the correctness policies that can be enforced with proactive libraries. Section 6 presents how to automatically generate proactive libraries from their specifications. Section 7 presents our empirical evaluation. Section 8 discusses related work. Section 9 provides final remarks.

2 BACKGROUND

This section provides background information about edit automata, which is the formalism we used to model the behaviour of the enforcers, the model-driven software development paradigm, which is the development methodology we used to define the modelling environment and the generators of the enforcers, and the Xposed framework, which is the framework we used to implement the proactive modules.

Edit Automata

Enforcers are software components that monitor other programs to check their correctness and modify their behaviour when problems are detected. We model the behaviour of the enforcers using edit automata, which are abstract machines that can specify how a sequence of operations executed by a monitored program must be transformed into an alternative sequence of operations [31]. In practice, we use edit automata to specify how to enforce behaviours that the monitored program may fail to satisfy.

More formally, an edit automaton is a finite state model with the capability to modify an input sequence of operations into an output sequence of operations by incrementally suppressing and inserting operations. Contrary to a normal finite state machine, the transitions of edit automata are annotated with both an input operation and an output operation sequence. According to the semantics of the transitions, when the input operation is detected, the output sequence is emitted. The output sequence might be *the same* as the input operation, which corresponds to not altering the execution, may contain *additional* operations, which corresponds to inserting additional operations in the execution, or may *omit* the input operation, which corresponds to suppressing the input operation.

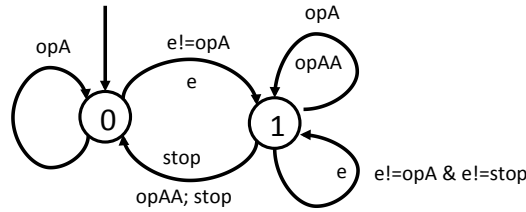


Fig. 1. A sample edit automaton.

Figure 1 shows a sample edit automaton. The symbol above each transition indicates the input operation recognized by the transition, while the sequence below a transition indicates the output sequence emitted by the automaton when the input sequence is recognized. The input symbol can also be expressed with guard conditions, for instance $e \neq \text{opA}$ indicates any operation different from opA . In this case e is like a variable whose value depends on the concrete input operation that is recognized by the transition. If the event e is also reported below the transition, the same event recognized in input is emitted in output. The arrow pointing at state 0 indicates the initial state.

The sample edit automaton suppresses every execution of opA that is initially recognized (see the self-loop on state 0). Once an event different than opA is detected, it moves to state 1 where it leaves the execution unaltered unless opA or stop are detected (see the lower self-loop on state 1). The occurrences of the operation opA are replaced with the occurrences of the operation opAA (see the upper self-loop on state 1). While when a stop operation is recognized, the edit automaton introduces an extra opAA operation before the stop operation and goes back to state 0.

For example, the edit automaton in Figure 1 transforms the input sequence
 $\text{opA}; \text{opA}; \text{opB}; \text{opA}; \text{opA}; \text{opC}; \text{stop}$

into the sequence

$\text{opB}; \text{opAA}; \text{opAA}; \text{opC}; \text{opAA}; \text{stop}$

When the edit automaton is used to specify how to enforce a policy for a given library, the input and output operations are events that correspond to method calls and to the termination of methods execution. These methods can belong to either the target library or to an object lifecycle (e.g., an activity).

Model-Driven Software Development

Model-Driven Software Development (MDSD) is an approach for developing software by extensively exploiting models [52]. MDSD extends the notion of models as a form of documentation, to consider them as artefacts equivalent to code, since they can be used to automatically generate executable code.

Code can be generated from models by applying a series of transformations, which can be organized in two logical sets: the implementation independent and the implementation dependent transformations. The implementation independent transformations are applied to an abstract model of the system to obtain a more concrete model that can be used as basis for code generation by applying the implementation dependent transformations.

The abstract model expresses the semantics of the system regardless of the software platform used for its implementation (e.g., Android, iOS, .NET, and so on). In general, the abstract model is at the same abstraction level of classic analysis models but represents the behavior of the system extensively.

To apply MDSD a number of tools are necessary, including modeling, model validation, and code generation tools. The modeling tools offer editors for the definition of the models using a domain specific modeling language based, for example, on UML profiles. Model validation tools verify the correctness of a given model with respect to a model domain-specific metamodel, which can be defined in a language such as OCL [42]. Code generation tools transform models into executable code from a given model. Code generation can only work if the model is correct with respect to the metamodel and therefore model validation is performed before code generation.

In our approach to the design of software enforcers, we implemented modeling tools to specify the enforcers, and defined both domain independent and domain dependent transformations to automatically generate the proactive modules that implement the behavior specified by the enforcers from the models.

Xposed

Xposed is a framework that can be installed in a rooted smartphone to allow users easily add modules that can customize the features present in the smartphone [56]. There exists a number of standard modules ready to be downloaded and installed.

In addition to using existing modules, it is possible to implement new modules while exploiting the core capabilities of the Xposed framework, which include the ability to capture and modify the execution flow of Android apps. This ability is achieved by intercepting class loading events. Every time a class is loaded, its code can be instrumented on the fly before it is executed. The implementation of the module must specify the code locations that must be instrumented and the code that must be inserted at these places, similarly to what is done with aspect oriented programming [26] frameworks. We exploit this framework to implement the software enforcers as Xposed modules.

3 RUNNING EXAMPLE

In this section, we present a motivating example based on an app for capturing pictures in Android, *Plumeria*, that we downloaded from git (<https://github.com/DonLiangGit/Plumeria>). We selected this app because it is simple, but yet complex enough to illustrate the key concepts related to proactive libraries.

Listing 1 shows a code-snippet of `MyActivity`, which is the Android activity that uses the `Camera` in *Plumeria*. On startup, the callback method `onCreate()` is executed and a new `Camera` object is created by method `getCamera()`, whose implementation acquires the backward camera of the device invoking the method `Camera.open()`. The method `Camera.open()` is an Android library method that can be used to request the access to the camera, while the corresponding library method for releasing the camera is `Camera.release()`.

The public list of issues for this app reports an incorrect use of the camera⁵. In fact, every time `MyActivity` becomes invisible, the camera becomes inaccessible to the other apps of the device. The camera is not even anymore accessible from `Plumeria` because when `MyActivity` is visible again, the `onCreate()` callback method is executed, and the call to `Camera.open()` produces an exception, since the app is attempting to acquire a camera that is already in use (by the app itself).

Listing 1. A code-snippet that shows the activity that handles the camera in `Plumeria`

```
...
public class MyActivity extends TabActivity {

    private Camera camInstance;
    ...

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my);

        camInstance = getCamera();
        cameraPreview = new CameraPreview(this, camInstance);
        preview = (FrameLayout) findViewById(R.id.camera_preview);
        preview.addView(cameraPreview);
        ...
    }

    public static Camera getCamera() {
        Camera cam = null;
        cam = Camera.open();
        return cam;
    }

    ...
    @Override
    public void onPause() {
        super.onPause();
    }
    ...
}
```

This fault is caused by the app that does not handle the access to the camera properly. In particular, the `onPause()` callback method, which is invoked when the activity becomes invisible, does not invoke `Camera.release()`, contrary to what is recommended in the Android API documentation⁶.

In the rest of the paper, we exploit this faulty app to present the concept of a proactive library and to show how the proactive module associated with the `Camera` API may automatically detect and heal this problem.

4 PROACTIVE LIBRARIES

A proactive library is a *reactive library* augmented with one or more *proactive modules*. The reactive library is a regular library that can be normally used as part of an application. In this paper we focus on Android libraries that control access to resources but the concept is generally valid for any library. The proactive modules are software

⁵See issue <https://github.com/DonLiangGit/Plumeria/issues/1>.

⁶See <https://developer.android.com/guide/topics/media/camera.html#release-camera>.

components that enforce *correctness policies* by altering executions, for instance by adding or suppressing method invocations.

Although the modules are reactive per se, that is, they affect the execution when specific events are detected, their behaviour is proactive with respect to the associated library. In fact, the proactive modules can modify the executions regardless of the behavior of the library. In particular, proactive modules react to the behaviour of apps and libraries. The behaviour of the apps is observed in terms of the callback methods that are executed. Invocations to *callback methods* are calls produced by the framework (in this case the Android framework) to methods implemented in the app for correctly handling lifecycle state transitions. For example, in Android there are methods that are automatically invoked by the framework when an activity is suspended, destroyed, resumed, and so on [3].

The behavior of the libraries is observed by observing calls to *library methods*, that is, calls to the methods implemented by the reactive library.

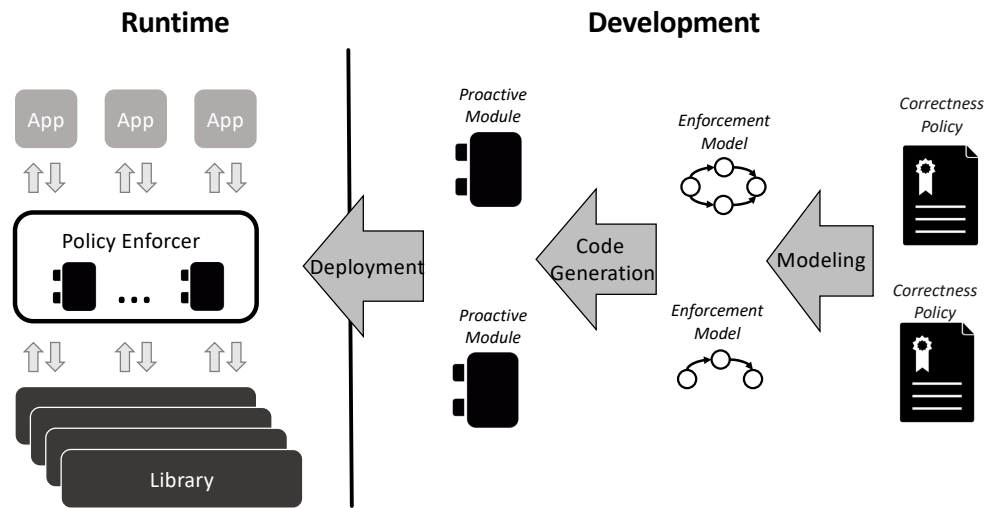


Fig. 2. Proactive Libraries.

Correctness policies specify how a library should be used by an app relating the status of the app, traced by observing calls to callback methods, to the status of the library, traced by observing calls to library methods. Correctness policies may potentially cover any aspect related to the usage of a library, including functional correctness, privacy, and security aspects. In this paper we focus on a specific class of correctness policies that we call *resource usage policies*. These are policies that state how the API interface of a library that controls the access to a resource should be used to prevent any misuse of the resource, which may cause misbehaviors, failures and crashes at runtime.

Note that in contrast to a regular reactive library, a proactive library may perform operations even when the library is not in use. For instance, the invocation of a callback method may trigger the execution of a proactive module, which may proactively perform some operations necessary to guarantee the correctness of the execution, regardless the direct involvement of the library in the computation. Considering the running example, a proactive module can fix the execution by forcing the release of the camera when `MyActivity` is no longer visible, that is, when the `onPause()` callback method is invoked, and re-assigning the camera to the app when the activity is resumed, that is, when the `onRestart()` callback method is invoked.

Figure 2 visually illustrates how proactive libraries work, distinguishing the development and the runtime phases. At development time, developers of proactive modules first identify the correctness policies that they want to enforce. A *correctness policy* is a natural language statement that constrains the usage of the API, considering the status of the app as captured by its lifecycle, if necessary. Any app that violates a correctness policy is faulty.

Examples of correctness policies, and more in particular resource usage policies since they refer to access to resources, for the Camera Android library are:

Resource Usage Policy 1: “*An activity that is paused while having the control of the camera must first release the camera.*”

Resource Usage Policy 2: “*An app cannot acquire the camera twice.*”

The first policy refers to both the status of the activity (an activity that is paused) and the usage of the API, while the second policy only refers to the usage of the API. Both policies are valid for every app that may use the camera. Note that the first policy is the one that should be exploited to heal the failure caused by the fault in the running example.

Correctness policies are then formalized and encoded as enforcement models. An *enforcement model* precisely defines how to react to any violation of the correctness policies. We use edit automata [31] to define the enforcement models because they represent a simple finite-state based formalization that naturally supports the specification of the behavior of the proactive module in terms of the events that must be intercepted and the events that must be inserted and suppressed as a reaction to each intercepted event. The objective of the enforcement model is to *enforce* a correctness policy, when the running apps do not satisfy it. The enforcement model is defined uniquely using the knowledge of the library API and the Android lifecycle events (i.e., the callback methods) [3], which are the same for any app. Thus its definition does not require any knowledge specific to the app that uses the library. See Section 5 for details about how to define enforcement models using edit automata.

An enforcement model fully describes the behavior of the corresponding proactive module, which can be thus generated from the model. Proactive modules can be deployed in any environment where the corresponding library is used. Since proactive modules are activated by the invocation of certain methods, their execution in the user environment is controlled by a *policy enforcer* that intercepts the events and dispatches them to the deployed proactive modules. The policy enforcer also controls the activation and deactivation of the proactive modules, which can be turned off and on by the user. See Section 6 for details about how to generate and execute a proactive module.

5 ENFORCEMENT MODELS

An enforcement model is a formal representation of the actions that must be undertaken to automatically enforce a correctness policy, that is, the operations that must be executed to turn an execution that violates a policy into an execution that satisfies it. In this paper, we focus on the Android ecosystem and on a specific class of correctness policies, the *resource usage policies*, which state how an API that controls the access to a resource must be used by apps. An interesting aspect about these policies is that the usage of a resource is strongly coupled with the status of apps and their components, that is, depending on the state of an Android component as captured by its lifecycle there are operations that must or must not be executed.

In practice, an enforcement model traces the state of the app by intercepting the system callbacks, for instance the system callbacks can be used to uniquely identify the state of an activity according to the Android activity lifecycle [3, 45], traces the state of the library by intercepting the calls to the library API methods, and specifies the actions that must be proactively and automatically suppressed or inserted to enforce the satisfaction of a possibly violated policy. The proactive module obtained from the enforcement model guarantees the satisfaction of the policy at runtime, so that executions can be healed automatically.

In the following, we discuss how edit automata can be used to specify enforcement models.

Policy Specification

An enforcement model formally defines the actions that must be automatically undertaken to enforce policies. In the case of resource usage policies for the Android ecosystem, the enforcement models can mainly suppress and insert actions of two kinds: callback methods and API methods. Callback methods are method calls produced by the Android framework when an app changes its status. For instance, the callback method `onStop()` is automatically invoked when the running activity of the app is no longer visible, while `onDestroy()` is invoked when the activity is destroyed. The API methods are the methods implemented by the library associated with the proactive module. In the running example, the methods `open()` and `release()` are API methods implemented by the Camera.

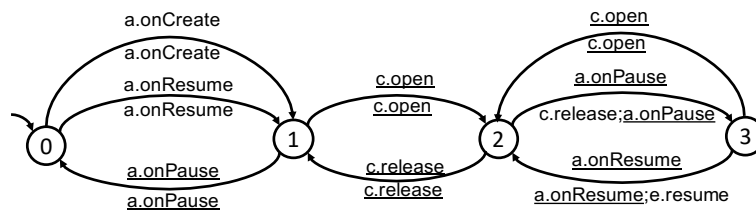


Fig. 3. Enforcement model for the Resource Usage Policy 1 of the Camera.

Figure 3 shows the enforcement model for the Resource Policy 1 of the Camera API reported in Section 4. Compared to general edit automata, our enforcement models extend the standard notation to support three aspects specific to our application context: prefixes, typed events, and special operations.

We use *prefixes* to disambiguate the element that performs the operation represented in the model. In this example, the prefix *a* represents the Activity class, the prefix *e* represents the class that implements auxiliary methods that can be invoked by the enforcer, and the prefix *c* represents the camera API. In the actual automata editor the full name of classes is used as prefix of methods.

The events in the model could be of two types: events that identify the beginning of an operation and events that identify the end of an operation. The type of an event can be distinguished based on its name: underlined event names correspond to the end of an operation, while regular names correspond to the beginning of an operation. For example, the outgoing transitions starting from state 0 in Figure 3 represent the beginning of the execution of the callback methods `onCreate` and `onResume`, while the transition from state 1 to state 2 represents the completion of the execution of method `open` implemented by the camera.

This distinction is important to precisely represent the events that must activate the enforcers. For instance, the enforcer must capture the beginning of `onCreate` and not its end to check if `open` is correctly invoked during the execution of this callback method, that is, after `onCreate` begins its execution but before it finishes it. Similarly, the enforcer must capture the conclusion of the callback method `onPause` from state 2 to detect the case the resource has not been released, even while executing `onPause`.

Finally, the model may include events that are neither callback methods nor API methods but whose semantics is known by the code generator. This capability is exploited to address cases that can be hardly managed within the enforcer but that can be easily addressed with appropriate tool support. These events are represented with the prefix *e*, to indicate enforcer methods, such as the event *e.resume* in Figure 3.

In this case, the event is used to activate a resume policy specific to the target resource. The policy could be in principle explicitly encoded in the model, but this would unnecessarily complicate the specification of

the enforcers. To address these cases, we rather extended the code generator with special operations of known implementation that can be embedded in the generated code, as discussed in Section 6. In the context considered in this paper, we used this capability only to support resume operations, since they may require the generation of multiple code fragments that trace the values of the variables used in the implementation of the resume procedure.

The example model in Figure 3 enforces the release of the `Camera` when the activity goes to background while it still has the camera. In the initial state (state 0), the model waits for the creation of a new activity. Once the activity has been created (state 1), the enforcer checks whether a camera is opened. If a camera is opened, the enforcer checks that it is released before the activity is paused (transitions from state 2 to state 1 and from state 1 to state 0). In all these cases the execution is unaltered because the app would be using the camera consistently with the policy that requires the camera to be released every time the activity is suspended.

However, the app might be paused without releasing the camera (transition from state 2 to state 3). In this case the enforcer changes the execution forcing the release of the camera. When the execution of the activity is recovered, the camera is reassigned back to the activity if the activity does not already take the camera back autonomously (transitions from state 3 to state 2), so that the behavior of the enforcer is fully transparent to the app.

Using edit automata to specify the behavior of a proactive module compared to directly coding the module has several advantages: (1) it provides a compact representation of the enforcement strategy that can be more easily manipulated and modified than working at the code level, (2) it can be used to automatically generate an implementation of the enforcement strategy, reducing the risk of introducing faults and speeding up the implementation process, and (3) it can be composed with other finite state specifications (e.g., the specification of other policies, the app lifecycle, and the protocol of the API) to formally check its correctness [47].

6 GENERATION OF PROACTIVE MODULES

A proactive module is a software component that verifies the correctness of the execution according to the strategy defined in an enforcement model. Since implementing proactive modules manually is an error prone and time consuming process, we defined a Model-Driven Software Development (MDSD) process and the corresponding tool chain to obtain proactive modules automatically from the enforcement models.

Our MDSD tool chain implements the core functionalities of a MDSD environment:

- *Modelling*, which provides the capability to model enforcement models that encode resource usage policies;
- *Validation*, which provides the capability to validate the syntactic correctness of an enforcement model;
- *Code generation*, which provides the capability to automatically generate the source code that implements the enforcement logic described by the enforcement model;
- *Compilation and deployment*, which provide the capability to compile and deploy the generated proactive module on a target platform.

We assigned these responsibilities to two main components: the *EMEditor*, which implements *modelling* and *validation* capabilities, and the *PMGenerator*, which implements *code generation*, *compilation*, and *deployment* capabilities.

Figure 4 shows the activities that a developer who uses our MDSD-based environment follows to define enforcement models and generate the proactive modules. The developer uses the *EMEditor* to visually design a model that enforces the considered policy. The output of the *EMEditor* is a validated enforcement model, which is the starting point for the generation of the proactive module. The developer then uses the *PMGenerator* to generate and deploy the proactive module corresponding to the input enforcement model. Code generation is based on two sets of rules that distinguish the platform independent and the platform dependent part of the process. In the following, we describe the *EMEditor* and the *PMGenerator*.

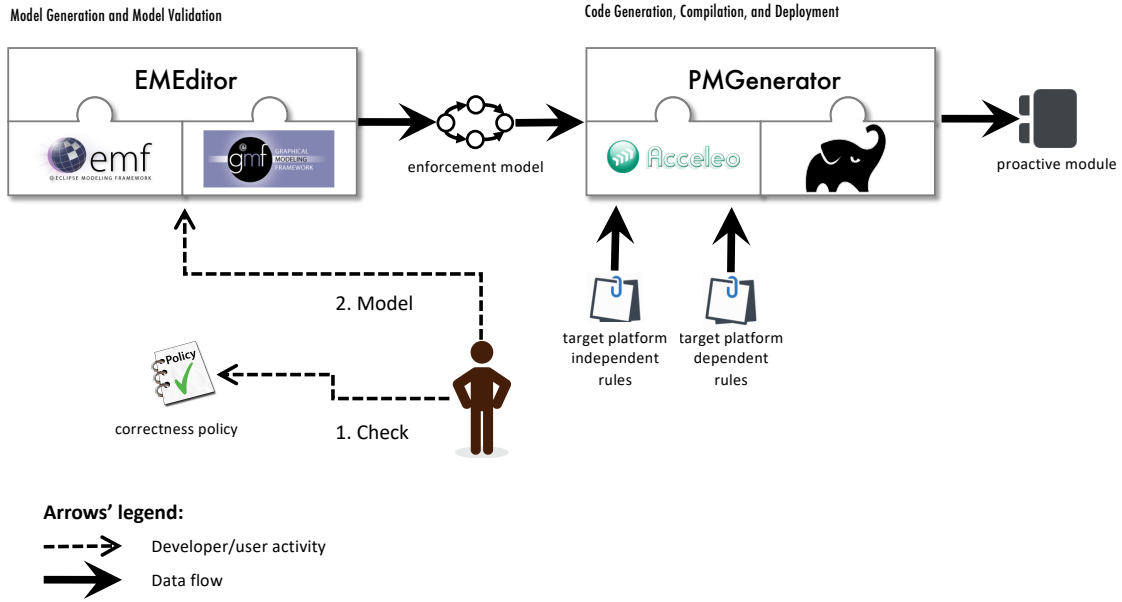


Fig. 4. The end-to-end tool chain of our MDSD-based environment

6.1 EMEditor

We implemented the EMEditor in Eclipse [18] using the Eclipse Modeling Framework (EMF) [19] and the Graphical Modeling Framework (GMF) [20] plugins. EMF can be used to create modeling and code generation tools starting from a data model. GMF can be used to implement graphical editors in Eclipse. We thus created the EMEditor by defining models in EMF and obtaining the corresponding graphical editors with GMF. Figure 5 shows the set of models that we defined and combined.

The *domain model* is the meta-model that defines what enforcement models are. When the developers use the EMEditor to define enforcement models they create instances of the classes defined in the domain model. Our domain model is shown in Figure 6.

The *EnforcerAutomaton* is characterized by a name, a *lifecycleObject* that uses the API, and the API that we want to be used correctly. The automaton is represented as a set of *States* connected by *Transitions*. A *State* could be an *InitialState* or a *RegularState*. *Transitions* are annotated with both the name of the operation that is intercepted, attribute *interceptedAction*, and the possibly empty sequence of actions that are generated by the enforcer, attribute *actionToPerform*. Those two attributes are associated with two type definitions that constrain the values of these strings according to a syntax that is checked by the editor. The regular expressions associated with type definitions are not shown in the model, however an *ActionSignature* is constrained to be a method name with a prefix that indicates whether the enforcer should intercept the execution of the method when it is invoked (prefix *before#*) or when it returns (prefix *after#*). The *ActionSequence* is constrained to be a comma separated sequence of action signatures.

We then derived the *domain generation model* from the domain model through a standard EMF model-to-model transformation. This model enriches the domain model with additional information useful to the editor, such as, the set of properties associated with each graphical element.

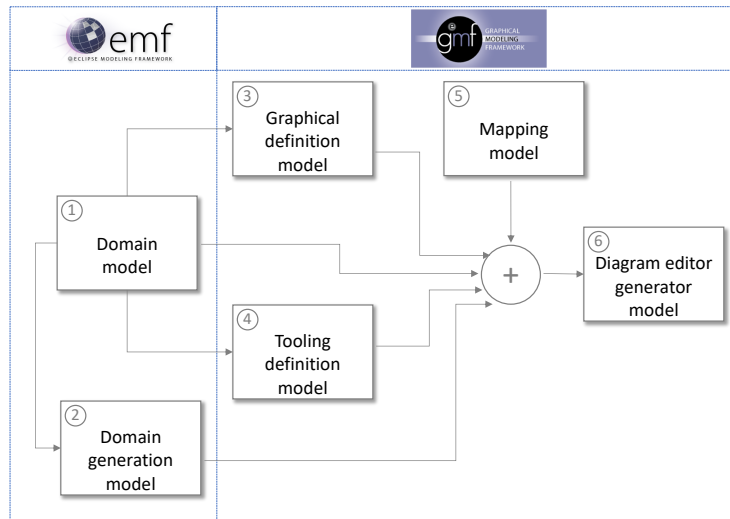


Fig. 5. The models and their composition that define the EMEditor

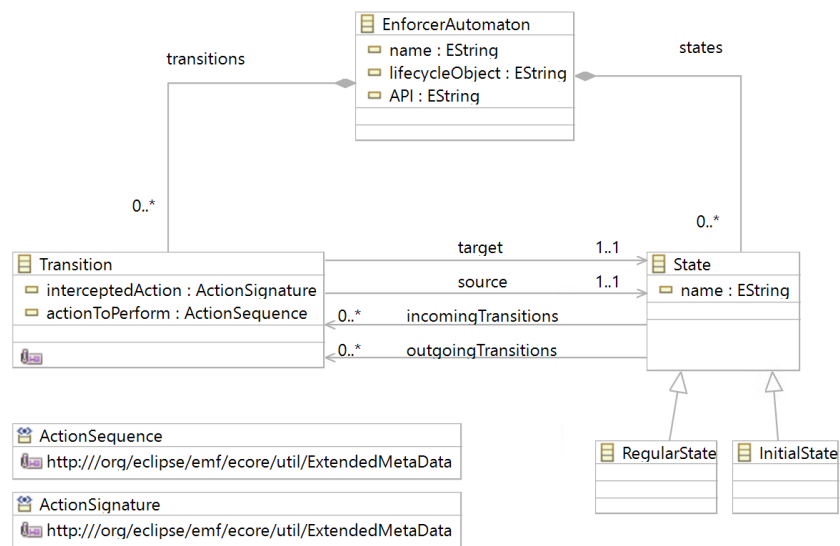


Fig. 6. The domain model of an enforcement model defined with the EMF framework

To obtain the visual part of the EMEditor we exploited GMF, which can be used to generate an editor where each entity of the model has a user-defined visualization. To define the editor with GMF we defined four models, as shown in Figure 5.

The *graphical definition model* associates a graphical representation with each element of the domain model that must be displayed within the drawing sheet of the editor. In our case, we defined shapes for all kinds of states and transitions.

The *tooling definition model* defines the palette and the other accessory tools, such as menus and toolbars. In our case, we defined a palette with four tools: three tools to create states of two types (initial and regular) and one tool to create transitions. We also included features provided by GMF, such as the ability to zoom in and out and the possibility to add notes.

The *mapping model* defines how to obtain the components of the editor, that is, the *diagram editor generator model*, by combining the domain model, the graphical definition model, the tooling definition model, and the domain generation model. In practice the mapping model mainly associates the elements in the domain model (i.e., states and transitions) with both the respective geometric shapes defined in the graphical definition model and the palette tools defined in the tooling definition model. The constraints defined in the domain model (e.g., the constraints on the annotations associated with transitions) are also propagated to the diagram editor generator model. The mapping model includes additional constraints that verify the integrity of the model drawn in the editor (e.g., the presence of an initial state and the existence of the Java methods specified in the model).

Finally, the EMEditor is automatically obtained from the diagram editor generator model using GMF, as shown in Figure 5. Figure 7 shows the enforcement model corresponding to the one defined in Figure 3 drawn in the EMEditor. The models can be saved as XML which are then processed by the PMGenerator.

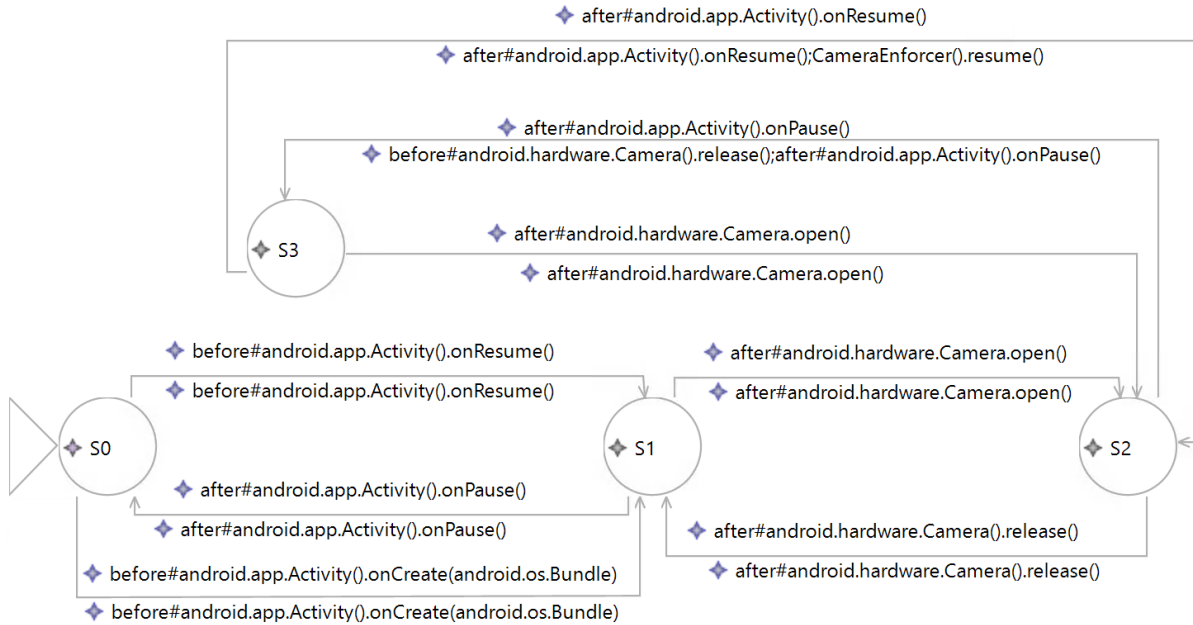


Fig. 7. The enforcer corresponding to the model in Figure 3 modeled with the EMEditor.

6.2 PMGenerator

The enforcement model defined in the EMEditor can be used to generate the actual proactive module that is deployed in the target environment to actuate the enforcement strategy. In this section, we first present the

architecture of the proactive module, and then we describe how the components of a proactive module can be generated automatically.

We designed the architecture of a proactive module to be generic, relying only on the possibility to implement two features with the selected technologies in the target environment: (1) the ability to intercept method calls, and (2) the ability to encapsulate the instance of the library controlled by the proactive module. These two features can be obtained in many different ways with different technologies and in different environments. The information about the specific technologies to use is encoded in the sets of rules used by the PMGenerator to emit the proactive module.

To assess our approach, we implemented the PMGenerator for Android using Xposed [56], which allows to cost-efficiently intercept method invocations and change the behavior of an Android app using run-time hooking and code injection mechanisms. Figure 8 graphically illustrates the architecture of a proactive module instantiated on Android.

Xposed transparently intercepts the method calls to the encapsulated resource and to the object life cycles and propagates them to the policy enforcer, which reacts based on the behavior defined in the enforcement model. For instance, it might introduce additional method calls or suppress calls.

Every interaction with the target library, including interactions originated from the app and from the policy enforcer, is mediated by a resource manager. In this way, the reference to the target library is stored inside the resource manager and it can be destroyed and recreated transparently to the app. This capability is sometime relevant to the policy enforcer, for instance when there is an instance that must be destroyed and recreated to release and acquire a resource. Similarly, the resource manager detects the identity of the Android component that interacts with a given library instance, preventing the application of the enforcement strategies to cases not supported. The resource manager is also injected in the application using Xposed to redirect method calls.

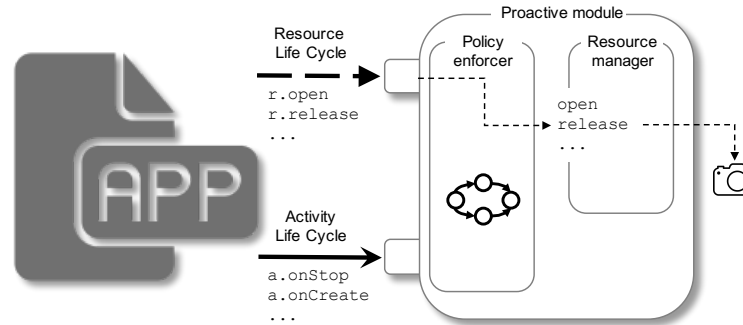


Fig. 8. The architecture of the proactive module

To automatically generate the proactive module from the model defined with the EMEditor tool, we used Eclipse [18] augmented with the Acceleo [17] plugin, which is an open-source code generator for model-driven development.

As shown in Figure 4, the code generation task uses two sets of rules that define how the resulting code should be structured, distinguishing between the rules that depend on the target platform (e.g., Android and Xposed) and the ones that are independent (e.g., rules that depend on the choice of the target language online).

The platform-independent rules map the elements in the enforcement model to an implementation in a given target language. In our case, we use the switch-case design pattern [24] to properly organize the generated code

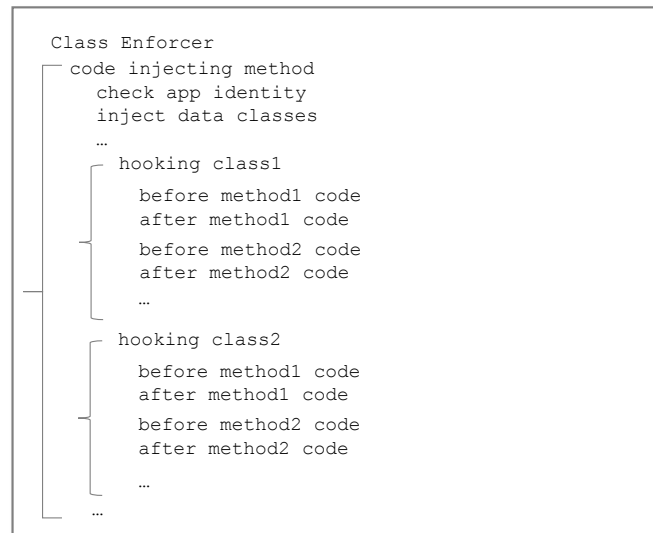


Fig. 9. Logical structure of the implementation of an enforcer for Android

in a Java class. The platform-dependent rules generate the additional code necessary to intercept method calls in the target environment (e.g., Android) with a given technology (e.g., Xposed).

Figure 9 shows the high-level structure of the file generated automatically for Android using the Xposed framework. Each enforcer is a class with a single method (the *code injecting method*) that implements the code injection logic. The method is invoked every time the device loads new packages. The initial part of the method (*check app identity*) checks if the loaded package belongs to an app that must be affected by the enforcer, otherwise it returns immediately. The rest of the code (*inject data classes*) injects auxiliary classes that might be used from the code injected afterward, for instance the implementation of the resume operation in the running example.

The code injecting method then includes a series of blocks that inject code in the individual classes that must be instrumented (*hooking class* methods). Each block looks for the class to instrument first, and then injects specific behaviours that are executed before (*before method code*) and after (*after method code*) the target method is executed. The injected code fragments are themselves defined as methods that are executed before or after the target method.

Listing 2 shows an excerpt of the code corresponding to the model in Figure 3 generated by the PMGenerator. The blue underlined code is the code obtained with platform independent rules.

The `handleLoadPackage()` method is the method responsible for instrumenting the loaded packages. The invocations to `findClass` search for the classes that are specified in the model and that must be instrumented. The `getAppClassesFromDex` extracts the classes in the application package. In the cycle that iterates over these classes, the code checks if the class in the app package is the same or a subclass of the classes that have to be instrumented. If the class has to be instrumented, the `findAndHookMethod` invocation identifies and instruments the target methods. For instance, the sample code instruments the `open` method of the `android.hardware.Camera` class and its subclasses, and the `onPause` method of the `android.app.Activity` class and its subclasses.

Listing 2. A code-snippet of the proactive module corresponding to the enforcement model in Figure 3

```

public class CameraReleaseEnforcer extends IXposedHookLoadPackage {
    ...
    public void handleLoadPackage(XC_LoadPackage.LoadPackageParam lpparam) throws Throwable {
        ...
        Class<?> apiClass = findClass("android.hardware.Camera", lpparam.classLoader);
        Class<?> lifeCycleObjectClass = findClass("android.app.Activity", lpparam.classLoader);
        ArrayList<Classes<?>> appClasses = getAppClassesFromDex(lpparam);
        ...
        for (Class<?> appClass : appClasses) {
            ...
            if(apiClass.isAssignableFrom(appClass)){
                findAndHookMethod(appClass, "open", new XC_MethodHook() {
                    ...
                    protected void afterHookedMethod(MethodHookParam param) throws Throwable {
                        ...
                        //transition from state 1 to state 2 (camera open)
                        if (S1.equals(currentStates.get(lifeCycleObject))) {
                            currentStates.put(lifeCycleObject, S2);
                        }
                        ...
                    }
                });
            }
            ...
        }
        ...
        if(lifeCycleObjectClass.isAssignableFrom(appClass)){
            findAndHookMethod(lifeCycleObjectClass, "onPause", new XC_MethodHook() {
                ...
                protected void afterHookedMethod(MethodHookParam param) throws Throwable {
                    ...
                    //transition from state 2 to 3 (activity on pause)
                    if (S2.equals(currentStates.get(lifeCycleObject))) {
                        doNotAlterateExecution = true;
                        Camera resource = lifeCycleObject2resource.get(lifeCycleObject);
                        saveCameraState(resource, lifeCycleObject);
                        resource.release();
                        doNotAlterateExecution = false;
                        currentStates.put(lifeCycleObject, S3);
                    }
                    ...
                }
            });
        }
        ...
    }
}

```


When a method is instrumented, the body of the method `findAndHookMethod` may contain the `beforeHookedMethod` and the `afterHookedMethod` inner methods, which define the operations that must be executed before and after the target method, respectively.

The structure of the enforcer discussed so far is the result of rules that depend on the target platform. The body of the `beforeHookedMethod` and `afterHookedMethod` however may derive from rules of both kinds.

The code that depends on the platform is the one that retrieves and stores variables from data structures to make them accessible by the code generated from the enforcement model, such as the code that retrieves the current state of the enforcer (`currentStates.get(lifeCycleObject)`) and the code that retrieves the resource associated with the current activity (`resource = lifeCycleObject2resource.get(lifeCycleObject)`). Note that the implementation stores and retrieves this information while taking into account the mapping between the activity, the resource manipulated by the activity, and the enforcer that controls that interaction (a different enforcer instance is created for each pair activity-resource).

The rest of the code is generated from the enforcement model only. The code that derives directly from the model could be as simple as changing the current state only, such as for the `afterHookMethod` of the `open` method, or it might be more complex. For instance, the `afterHookMethod` associated with the `onPause` method generates an additional method call (`resource.release`) to force the release of the camera. The code also includes statements to avoid that the enforcer intercepts the calls produced by the enforcer itself (see usage of variable `doNotAlterExecution`) and code to store variable values to later enable the possibility to resume the resource.

Once the file has been generated, it can be compiled and deployed on the target platform. Our MDSD-environment exploits Gradle to import the generated source file in an Xposed project defined in Android Studio IDE [5]. Our prototype implementation of the tool can be downloaded from <https://gitlab.com/learnERC/proactivelibrary>.

Note that the entire process is independent on the apps that will be affected by the enforcers. In fact, our MDSD-environment does not require to know the identity of the apps that may violate a policy but only the library API involved in the policy that must be enforced. The target apps are identified by the user when enabling the enforcement mechanism for one or more target apps. The enforcement mechanism checks the identity of the app in the *check app identity* step before the rest of the enforcement logic is applied. If the app is not among the ones to be enforced, its behavior is not modified.

7 EVALUATION

To evaluate proactive libraries, we considered three research questions, one about the effectiveness, one about efficiency, and one about the usefulness of the MDSD environment.

RQ1 (effectiveness): Can proactive libraries be used to detect and heal actual library misuses in Android applications?

With this research question, we evaluate the effectiveness of proactive libraries against actual problems reported in Android apps. As discussed in the paper, we focus on the misuses of the APIs that control the access to local resources because these misuses are quite frequent [6, 34, 55], and may have a significant impact on the system, for instance affecting the other non-faulty apps that run in the same device and use the same resources.

RQ2 (efficiency): What is the overhead introduced by proactive libraries compared to traditional reactive libraries?

With this research question, we measure the runtime overhead that can be experienced when the proactive modules are active compared to the standard configuration that executes the reactive libraries only.

RQ3 (usefulness): Can developers conveniently obtain proactive modules using the MDSD environment?

With this research question, we compare the effort needed to obtain proactive modules using the MDSD environment that we defined to the effort needed to directly implement them in Java.

Table 1. Apps

App Name	Version	Type	Downloads	Last Update	Links
AndroidHacks	N/A	open source	-	2018	https://github.com/YingVickyCao/AndroidHacks/
BlueChat	1.1	open source	-	2016	https://f-droid.org/wiki/page/com.alexkang.bluechat
CheDengWo	2.4.4	closed source	2.3 millions	2018	https://www.qimai.cn/andapp/downTotal/appid/4676
ErWeiMaL	1.0	closed source	4.2 millions	2015	http://apps.wandoujia.com/apps/com.weilaifu.qrcode
FontMaster	4.3.9	closed source	894 millions	2018	https://www.wandoujia.com/apps/com.zhiqupk.ziti https://www.qimai.cn/andapp/downTotal/appid/607
Foocam	1.2	open source	100+	2017	https://play.google.com/store/apps/details?id=net.phunehehe.foocam2
FromCat	580	closed source	20,000	2016	https://www.qimai.cn/andapp/downTotal/appid/143359
GetBackGPS	0.4.1	open source	-	2019	https://ruleant.github.io/getback_gps/
IPST2	N/A	open source	-	2017	https://github.com/fRueD4096/IPST2
MaMa	1.0	closed source	10,000+	2015	https://www.qimai.cn/andapp/downTotal/appid/881807
QiCaiScan	1.0	closed source	19,300+	2018	https://www.qimai.cn/andapp/downTotal/appid/151774
SuperTorch	3.2	closed source	17.9 millions	2015	https://www.wandoujia.com/apps/com.idiantech.torch https://www.qimai.cn/andapp/downTotal/appid/25771
WebPCsuite	3.1.6	closed source	1.5 millions	2016	https://www.appsapk.com/web-pc-suite/ https://apk-dl.com/web-pc-suite-file-transfer/com.geeksoft.wps
WiFiSaver	N/A	open source	-	2018	https://github.com/FilipBanak/WifiSaver
XiaoMiWifi	1.1.755	closed source	32.9 millions	2018	https://play.google.com/store/apps/details?id=com.xiaomi.router https://www.qimai.cn/andapp/downTotal/appid/1463

To investigate these three research questions, we selected a set of Android apps that have been reported to be affected by API misuses. In particular, we selected all the apps from the benchmark that Liu et al. [34] used to evaluate their static analysis technique for the detection of resource leaks. This consists of 9 closed-source apps and 3 open-source apps. We extended this set with 3 open source apps from GitHub for a total of 15 real-world Android apps. Table 1 lists all the apps, reporting their name (column *App Name*), version name when available (column *Version*), the type of app (column *Type*), the number of downloads of the app for the ones available on a store (column *Downloads*), the year last time the app has been updated either on the open source repository or on its online store (column *Last Update*), and the link to download the app from either the open source repository or the app store (column *Links*). The apps are from a variety of domains, including communication and productivity, are quite popular, and still maintained. All these apps use resources and have been reported to be potentially affected by a total of 27 resource misuses spanning 12 different resources, including the Media player, the Camera, and the Location manager.

For each case, we used Appium [21], a test automation tool for mobile apps, to manually implement an automatic test case that reproduces the sequence of actions that has been reported to produce the misuse. All the executions have been performed on an actual smartphone: the Samsung Galaxy S5 smartphone running the Android 6.0.1 Marshmallow mobile platform, equipped with a 2.5 GHz quad-core Snapdragon 801 system-on-chip, 2 GB of RAM, and 16 GB of internal storage.

RQ1 - Effectiveness

To evaluate the effectiveness of proactive libraries, we first identified the usage policies relevant to the APIs involved in the reported misuses and we then defined the corresponding enforcement strategies. To identify the policies, we exploited the information about the use of the Android API reported in [4, 34, 55]. Each enforcement

Table 2. Effectiveness of Proactive Libraries

APP	Lifecycle Object	API	Usage Policy	Policy Violation
AndroidHacks	Activity	android.app.Activity	If managedQuery() is invoked, replace it with query() of ContentResolver	healed
BlueChat	Activity	android.bluetooth.BluetoothAdapter	If enable() is invoked, invoke disable() when onDestroy()	healed
CheDengWo	Activity	android.media.MediaPlayer	If <init> is invoked, invoke release() when onPause()	healed
ErWeiMaL	Activity	android.media.MediaPlayer	If <init> is invoked, invoke release() when onPause()	healed
FontMaster	Activity	android.util.LruCache	If <init> is invoked, invoke evictAll() when onDestroy()	healed
Foocam	Activity	android.hardware.Camera	If open() is invoked, invoke release() when onPause()	no violation
	Activity	android.hardware.Camera	If startPreview() is invoked, invoke stopPreview() when onPause()	healed
FromCat	Activity	android.media.MediaPlayer	If create() is invoked, invoke release() when onPause()	healed
	MediaRecorder	android.hardware.Camera	If MediaRecorder.start() is invoked, do not invoke lock()	no violation
GetBackGPS	Activity	android.hardware.SensorManager	If registerListener() is invoked, invoke unregisterListener() when onPause()	no violation
	Activity	android.location.LocationManager	If requestLocationUpdates() is invoked, invoke removeUpdates() when onPause()	no violation
	Activity	android.os.RemoteCallbackList	If register() is invoked, invoke unregister() when onPause()	no violation
	Service	android.hardware.SensorManager	If registerListener() is invoked, invoke unregisterListener() when onDestroy()	no violation
	Service	android.location.LocationManager	If requestLocationUpdates() is invoked, invoke removeUpdates() when onDestroy()	no violation
	Service	android.os.RemoteCallbackList	If register() is invoked, invoke unregister() when onDestroy()	no violation
IPST2	Resources	android.content.res.Resources	If getDrawable is invoked, replace it with getDrawable of AppCompatActivity	healed
MaMa	Activity	android.media.MediaPlayer	If <init> is invoked, invoke release() when onPause()	healed
QiCaiScan	Activity	android.hardware.Camera	If open() is invoked, invoke release() when onPause()	healed
	Activity	android.hardware.Camera	If startPreview() is invoked, invoke release() when onPause()	healed
	Activity	android.media.MediaPlayer	If <init> is invoked, invoke release() when onPause()	no violation
SuperTorch	Activity	android.media.MediaPlayer	If <init> is invoked, invoke release() when onPause()	healed
	Activity	android.location.LocationManager	If requestLocationUpdates() is invoked, invoke removeUpdates() when onPause()	healed
WebPCSuite	Activity	android.media.MediaPlayer	If <init> is invoked, invoke release() when onPause()	healed
	Activity	android.os.PowerManager.WakeLock	If acquire() is invoked, invoke release() when onPause()	healed
WiFiSaver	Service	java.lang.Thread	If <init> is invoked, invoke interrupt() when onDestroy	healed
XiaoMiWiFi	Activity	android.net.wifi.WifiManager.MulticastLock	If acquire() is invoked, invoke release() when onDestroy()	no violation
	Service	android.net.wifi.WifiManager.MulticastLock	If acquire() is invoked, invoke release() when onDestroy()	healed

strategy has been codified as an edit automaton using the EMEditor and finally turned into a Java proactive module with the PMGenerator. The code generated for the proactive modules ranged from 1K lines of code in the case of `android.os.PowerManager.WakeLock` to 2K lines of code in the case of `android.media.MediaPlayer`. Quantitatively most of the code has been generated by the platform dependent rules.

To study the effectiveness of the proactive modules, we executed the test cases that should produce the misuses, and checked whether the misuses have been automatically detected and healed by the proactive modules. Table 2 reports the results. Column *APP* indicates the name of the Android app used in the evaluation. Columns *Lifecycle Object* and *API* indicate the Android component that has been reported to misuse the API and the API that has been reported to be misused, respectively. Note that each API maps to a different local resource.

Column *Usage Policy* indicates the policy that has been reported to be violated by the app. We have written the policies in the forms “if `methodA` is invoked, invoke `methodB` when `callback`”, “if `methodA` is invoked, replace it with `methodB`”, and “if `methodA` is invoked, do not invoke `methodB`”. The first policy should be interpreted as: if the app invokes `methodA`, it should also invoke `methodB` when `callback` is produced by the Android framework, unless `methodB` has been already invoked before. This policy requires interrupting any ongoing usage of a resource if certain events occur, such as the suspension or destruction of the current activity. The second policy should be interpreted as: if the app invokes `methodA`, the call should be replaced with a call to `methodB`. This policy requires replacing calls to faulty or deprecated methods with calls to methods that properly implement the required logic. The third policy should be interpreted as: if the app invokes `methodB` after `methodA`, suppress the call to `methodB`. This policy prevents the violation of API usage protocols.

Column *Policy Violations* indicates the effect of the proactive modules on the execution: *healed* indicates that the execution has been automatically corrected, while *no violation* indicates that the policy has not been violated by the execution. In none of the cases the proactive modules fail to heal an execution that violates a resource usage policy. This result confirms the effectiveness and suitability of proactive modules to enforce correctness policies.

It is worth discussing the ten cases where no violation has been detected. In the case of the *fooCam* app, the camera release policy has been reported as violated in [34], where a static analysis tool is exploited to detect erroneous accesses to resources. We discovered that this case in reality is a false positive produced by static analysis. The path that may violate the resource usage policy is infeasible, thus it can be never executed. This case is correctly captured by the enforcement model that does not report any violation when the app is executed, contrary to static analysis. Similarly, *FromCat* and *QiCaiScan* apps happened to use the camera properly.

In the cases belonging to *GetBackGPS* and *XiaoMiWiFi*, which were also reported as violating the corresponding resource usage policies in [34], we found out that they are not actual policy violations. In these four cases Android activities do not violate any policy because the resources are in reality acquired and released by Android services, which are usually used to perform long-term operations in background. Enforcing the release of the resources in these situations would produce misbehaviours since resources would be released while they are still in use by services. Our proactive modules for activities correctly captured this situation. To confirm this result we also generated the proactive modules checking that Android services acquire and release the resources properly and no violation has been reported.

RQ2 - Overhead

To measure overhead, we measured the time spent to run each test case, when proactive modules are active and when they are not. In the case of apps with multiple policies to be enforced, the experiment was conducted by activating all the proactive modules applicable to the app. For example, in the case of *Foocam*, there are two modules that act at the same time when the activity is paused: one to enforce the release of the camera, and the other to enforce stop capturing and drawing preview frames.

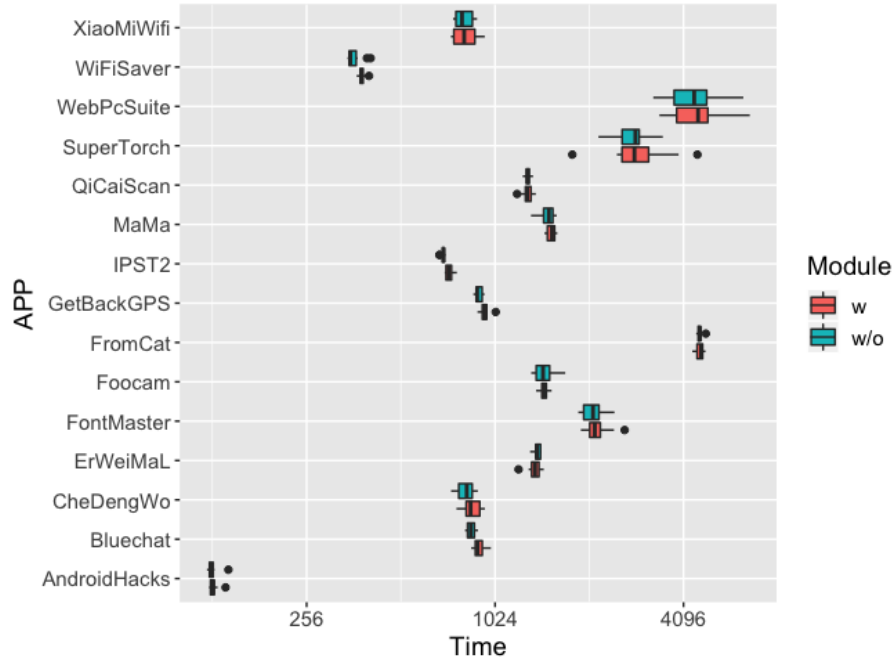


Fig. 10. Distribution of execution time w and w/o proactive modules

In addition to study the scalability of the approach, we measured how the overhead increases for an increasing number of monitored events. For this task, we selected the app whose proactive modules introduced the highest overhead, that is *Bluechat*, and we deployed an increasing number of enforcers. We measured the overhead for every configuration and report the results.

Figure 10 shows the overhead measured when the proactive modules that enforce the policies listed in Table 2 are active with a box plot. Each box represents a population of 10 samples obtained by running the same test case 10 times. We put next to each other the two sets of executions that we collected: the box labeled with w/o shows the runtime cost for the samples collected without activating the proactive module, while the box labeled with w shows the runtime cost for the samples collected when the proactive module is active.

We can notice that in most cases the use of proactive libraries does not generate any measurable overhead. This is confirmed with an ANOVA test that only reports the app as a significant factor ($p\text{-value} = 2e^{-16}$) and reports the presence of the proactive module as largely non-significant ($p\text{-value} = 0.99$). Overhead is low because the enforcement models for the majority of the cases leave the input unaltered or perform very simple operations. Considering all 15 apps we have an average overhead of 2%, even if in several cases (7 out of 15 apps) we have multiple modules active at the same time: 2 proactive modules for *Foocam*, *FromCat*, *SuperTorch*, *WebPCSuite*, and *XiaoMiWifi*, 3 proactive modules for *QiCaiScan*, and 6 proactive modules for *GetBackGPS*.

In the worst case we reported an overhead equals to 6% (the actual overhead was 52ms in the context of a functionality that required 858ms to complete), which is still an amount of overhead that can be hardly perceived [10]. The fact that the overhead introduced with the proactive modules does not introduce significant differences in the execution time of the apps has been confirmed with an ANOVA test that revealed no significant differences (with a significance level equals to 0.05) between the app with and without the enforcers running.

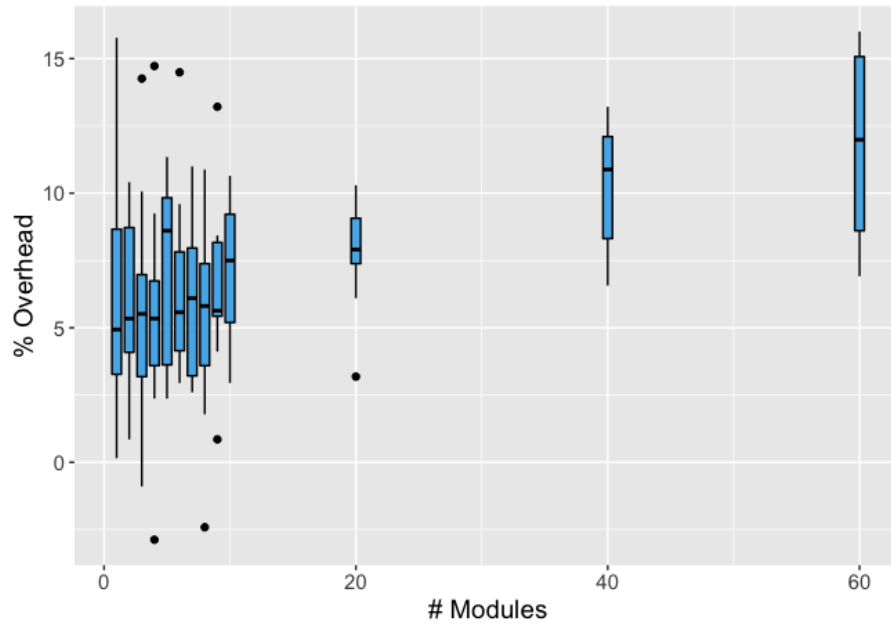


Fig. 11. Overhead vs number of proactive modules

Figure 11 shows the overhead (y axis) observed for an increasing number of proactive modules deployed on the app (x axis). Each box is the result of 10 executions. It is possible to notice how the approach scales well with the deployment of multiple proactive modules and thus the monitoring of many events. The presence of the monitoring infrastructure causes a mean overhead around 5% already when a single proactive module is deployed. The overhead however grows smoothly with the number of active modules. Indeed it is only slightly higher when 60 enforcers have been deployed reaching a mean value around 12%. This result shows that the approach does not suffer any particular limitation in terms of the number of enforcement strategies that can be simultaneously active.

Based on these results, we can conclude that enforcers can be feasibly used by final users to improve the reliability of their environment.

RQ3 - Usefulness

To investigate if the modeling and code generation environment is actually useful compared to implementing Xposed modules from scratch we defined an experiment involving human subjects. In particular, we involved three developers who graduated from our department with different levels of experience with Android and Xposed: an *Experienced User* with more than 1 year of experience with both Android and Xposed, a *Moderately Experienced User* with some months of expertise with both Android and Xposed, finally a *Novice to Xposed* user with years of expertise of Android development but only a 15 minutes training on Xposed.

After a 15mins debriefing on edit automata and our modeling environment, we asked each subject for each app to first design the enforcement strategy in our MDSD environment and then implement the same strategy with Xposed. We fixed a time limit of 3 hours for each task. Note that this setting is very favourable to the implementation of the modules directly with Xposed because the design effort necessary to identify the right

Table 3. Effort comparison

Subject	App	Lifecycle Object	Policy	Time with MDSD	Time with Xposed
<i>Experienced User</i>	SuperTorch	Activity	If <init> is invoked, invoke release() when onPause	15 min	2 h 51 min
	WiFiSaver	Service	If <init> is invoked, invoke interrupt() when onDestroy	16 min	2 h 25 min
<i>Moderately Experienced User</i>	SuperTorch	Activity	If <init> is invoked, invoke release() when onPause	32 min	>3h
	WiFiSaver	Service	If <init> is invoked, invoke interrupt() when onDestroy	25 min	>3h
<i>Novice to Xposed</i>	SuperTorch	Activity	If <init> is invoked, invoke release() when onPause	45 min	>3h
	WiFiSaver	Service	If <init> is invoked, invoke interrupt() when onDestroy	15 min	>3h

strategy is entirely spent in the modeling phase and when using Xposed the subjects already know the enforcement strategy they have to implement. Each subject worked on two apps and two policies, one policy that refers to Android activities and one policy that refers to Android services. To prevent human subjects from being tired while performing their tasks, the four tasks have been performed over four days, one task per day. We measure the time required to complete the task in the two cases for each subject. Finally, note that the human subjects had to use the actual apps only to validate their implementation, which can be anyway designed and implemented (generated in the case of our MDSD environment) regardless of the target app. In fact, the enforcement strategies require no information about the target apps that may violate the policies.

Table 3 reports the results. When using our MDSD environment, all the subjects managed to obtain correct enforcers in less than 1 hour, with the experienced user completing the task faster than all the other subjects. On the contrary only the experienced user managed to implement correct enforcers in less than three hours when using Xposed. This is mainly due to the complexity of the interactions between the monitored components, which is well abstracted in our modeling environment. In fact, the implementation has to deal with aspects such as tracing the identity of the components that interact with the considered APIs, transparently resuming resources that have been released, and avoiding the interference between the activity of the enforcer and the Xposed module. We can finally notice that although the experienced user finally obtained the module programmatically, he needed an order of magnitude more time than the time needed using our MDSD environment to achieve it. These results suggest that our MDSD environment provides a proper abstraction of the problem of designing an enforcement strategy and it can be useful to experienced users, to quickly obtain enforcers that can be deployed, and to both moderately experienced and novice users, who can more easily approach the technology.

Threats to Validity

The main internal threats to validity of our experiments relate to the definition of the enforcement model. In RQ1 and RQ2 we studied the applicability of proactive libraries using enforcement models designed by us. Although the enforcement strategies that we used are not particularly sophisticated at the model level, different users may design different strategies. To mitigate this issue we investigated the design of enforcement strategies with RQ3. All the subjects produced the same enforcement strategy consistently with the one designed by us when using our MDSD tool. In the case of the overhead, we further mitigated the issue by studying the scalability of the approach with respect to a number of configurations.

The investigation of RQ3 required the involvement of developers. To mitigate the issue about the background knowledge of the developers, we controlled this factor explicitly. To derive a worst case result, we also designed our study such a way that the developers implementing the Xposed modules already know how the enforcer should behave based on their experience with the MDSD tool. In reality the gap between subjects who use our modelling environment and subjects who implement modules directly would be higher.

The main external threat to validity is about the generalizability of the results. We considered various apps, faults, and strategies to mitigate this issue. The results that we achieved are quite consistent across cases so far, thus we expect a good degree of generalization, while we cannot exclude the existence of cases that can challenge our approach, as discussed in the next section.

Discussion

Proactive libraries can be a useful technology to deal with library misuses. In fact, if final users notice any misbehaving app, they can activate the available proactive modules to increase the reliability of their apps at the cost of introducing little overhead, as shown in our experiments. Note that these bugs may require a significant amount of time to be fixed. Out of the three bugs affecting open source apps, two of these bugs are still open at the time the paper was submitted and one bug has been fixed after 2 years and 3 months.

The evaluation confirmed the generality of proactive modules. In fact, enforcement strategies are defined using app-independent information only: information about the life-cycle of Android components and information about the library APIs. As a consequence, a same policy is always addressed with a same proactive module, which is reused across multiple apps. This is exactly what happened in our evaluation where we addressed the 27 cases reported in Table 2 with only 19 proactive modules, always reusing a same module across different apps when a same policy has been addressed.

Indeed proactive modules cannot be used to address any possible problem present in mobile applications. A first limitation is the direct consequence of their quality of being app independent. Since a proactive module relies uniquely on the library API and the app lifecycle events it can be applied to any Android app. On the other hand, app independence has a cost, that is, proactive modules cannot be used to deal with problems that require the knowledge of the app to be fixed, such as faults that do not imply any interaction with libraries but only depend on the internal behavior of the app.

Proactive libraries, as demonstrated in the empirical evaluation, can deal with multiple Android components, such as activities and services. The approach is not conceptually limited to these two components, but it can deal with any component and class interacting with a library. A limitation of the current implementation of our MDSD tool is that a same enforcement strategy cannot include multiple clients for a same library API. This limitation can be eliminated by extending the language of the edit automata and the corresponding tool support.

Another source of limitation is the need of intercepting events at run-time to inject and suppress events. In our prototype implementation we used Xposed, which requires rooting the device. This solution is not always practical because users might be unhappy with this requirement. However, there are also other options to affect the run-time behavior of apps. For example, apps can be instrumented in the smartphone, so that they can be monitored and enforced without requiring rooting the device, as done in the work by Neisse et al. [41]. We are considering this alternative implementation also for proactive modules.

Some events might be particularly hard to detect. For instance, peculiar app implementations such as the case of apps that bypass a library API producing native calls would not be intercepted by a proactive module. In general, the events that can be handled by proactive modules are either Java API calls or lifecycle callback events.

8 RELATED WORK

Android is one of the most popular software ecosystems that allows developers to easily develop and distribute apps, and allows users to easily retrieve and install applications. However, the development of reliable apps is a complex task that has to face several challenges, such as the unpredictability and the high variability of the environment [1, 39], the instability and rapid evolution of the software platform [32, 38], as well as the lack of strong and reliable analyses and testing tools [23, 25]. This often results in poorly developed applications, which may cause other apps or even the entire system to perform badly and crash.

In this paper we presented an approach that allows users to protect their environment from unreliable apps that improperly use APIs by enforcing the correct usage protocols. There are three distinct areas of research that are related to our contribution: detection of API misuses, incorrect resource management, and design of self-healing systems.

API misuses. Failures caused by API misuses are very popular [2]. These failures have been proven to be pervasive in software systems and different automated approaches are available to detect different types of misuses [30, 37, 53]. However, it is still impossible to completely prevent API misuses, due to limitations of the available techniques that can only detect specific classes of faults.

The presence of applications that incorrectly interact with an API could be further amplified by the rapid evolution of APIs, in fact developers often struggle to adapt their software to the latest versions of libraries. These problems are well-known to affect modern ecosystems, such as Android [14, 33]. The likelihood of encountering any of these problems while using an app downloaded from a marketplace is further increased by the lack of control over the process used to develop the apps available on the market. In fact, contributors are often more focused on rapidly prototyping their apps than developing high-quality applications. In a nutshell, despite the existence of techniques to assess interactions with APIs, problems due to API misuse are still often encountered by users [6, 7, 51, 54, 55].

This paper proposes to increase the reliability of devices that use applications developed by potentially third parties replacing traditional reactive libraries with proactive libraries, which can still be executed as reactive libraries when needed. Proactive libraries can transform the interaction between an API and unsafe applications to ensure that they do not violate usage protocols. This strategy can be effective in preventing multiple classes of problems, especially those related to the use of resources.

The proactive libraries share the idea of extending libraries with ReBa [13], which is a technique for the development of libraries augmented with adapters to ensure backward compatibility. However, ReBa and the proactive libraries have different goals and adopt different solutions. ReBa addresses the problems introduced by software evolution, while proactive libraries are a general mechanism for enforcing policies at runtime. In addition, ReBa is purely reactive, while proactive libraries use proactive modules to timely take actions to ensure correctness. Compared to a purely reactive solution, proactive modules can change executions more broadly by accessing information that otherwise cannot be intercepted.

Bad resource management. Although the concept of proactive library is general, many of the cases studied in the paper are about failures caused by bad resource management, such as apps that acquire and release resources according to wrong patterns [6, 28, 34, 46, 55]. Although some of these problems may be discovered with ad-hoc tests and static analysis techniques [6, 34, 55], it is generally difficult to eliminate resource management problems, covering every possible situation, also considering the problem of fragmentation that affects Android [54]. In addition, installing apps that misuse resources can also impact apps that properly interact with the same resources, increasing the need for solutions that can operate in the device. Our empirical results suggest that the reliability of interaction with resources can significantly improve if proactive libraries are adopted to replace reactive libraries, thus providing to library developers and app users a solution that complements the in-house testing and verification activities.

Self-healing. Techniques for avoiding and mitigating the impact of failures have been studied in many different contexts, including Web applications [35], cyber-physical systems [50], and Cloud computing [11]. Some techniques have also explored the idea of detecting and reacting to wrong situations, for example by using exception handlers [8, 9] and adapters [12]. However, only a few self-healing solutions have been designed to address an environment with limited resources such as a mobile device.

The first results in the Android domain were mainly related to dynamic patch injection, automatic suppression of faulty functionality, and healing of data loss problems. Dynamic patch injection can be used to quickly deploy fixes in apps [40, 57]. However, since patches are produced offline, healing is only achieved with the intervention of the developer, similarly to program repair techniques [22]. This mechanism can be useful to solve important vulnerabilities, but it cannot be used for immediate and automatic healing of failing executions.

Automatic suppression mechanisms can be used to automatically detect crashes and avoid future occurrences of the same crashes by bypassing the execution of features that caused the crash. [6]. This approach may be useful in preventing further failures, but does not help with fixing problems.

Finally, healing of data loss problems provides a strategy to prevent any data loss due to an incorrect implementation of mechanisms to suspend and recover the execution of the apps [43, 48].

Compared to these techniques, proactive libraries provide a design solution that is complementary to mechanisms such as dynamic patch injection, and potentially more general than approaches that address specific classes of failures, such as data loss problems.

It is worth mentioning that runtime enforcement in the Android environment has also been investigated by Falcone et al. [16], but their approach does not take into account the idea of designing policy enforcers that exploit the knowledge of general lifecycle events to increase the effectiveness of the enforcement strategies and does not include automatic code generation.

Finally, there are different approaches for the automatic adaptation of the software running on mobile devices to a changing context [15, 36, 49]. Although adaptation mechanisms are usually driven by the need for optimization rather than failure, it might be possible to explore how to apply proactive libraries to enforce adaptation policies rather than correctness policies.

9 CONCLUSIONS

Program libraries implement reusable functionalities that can be conveniently integrated into many different applications. The correctness of this integration depends on the ability to satisfy the assumptions of the library, usually consisting of constraints on when and how certain library operations can be performed. Unfortunately, satisfying these constraints is not always easy. This might be due to the complexity of the APIs implemented by the library, tacit assumptions about the library, and the complexity of the applications that use the library. For example, Android apps are particularly prone to incorrect interactions with their libraries [6, 34, 54, 55].

To address this problem, we introduced the concept of proactive library, which combines a regular reactive library with multiple proactive modules that can monitor executions and enforce correctness policies. These policies define how a library should be properly used at runtime. Since proactive modules are monitors that can alter executions according to a strategy, we propose to model their behavior as enforcement models, which offer a natural formalism to represent how an execution can be modified by suppressing and inserting method calls.

We use a model-driven software development process to automatically generate the proactive modules from models. In particular, we defined a code generation strategy that separates the code generated directly from the model and the code that depends on the target platform. This allows the code generator to be extended to other target environments while preserving the code generation rules that do not depend on the target platform.

We evaluated our solution on the Android environment focusing on relevant classes of APIs and correctness policies about access to resources. Results show that proactive libraries can effectively and efficiently enforce the specified resource usage policies.

ACKNOWLEDGMENT

The authors want to thank Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang for sharing with us information about the experimental subjects used to evaluate RelFix and for answering our questions.

This work has been partially supported by the H2020 Learn project, which has been funded under the ERC Consolidator Grant 2014 program (ERC Grant Agreement n.646867) and the GAUSS national research project, which has been funded by the MIUR under the PRIN 2015 program (Contract 2015KWREMX).

REFERENCES

- [1] D. Amalfitano, A. R. Fasolino, P. Tramontana, and N. Amatucci. 2013. Considering Context Events in Event-Based Testing of Mobile Applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.
- [2] S. Amani, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini. 2016. MUBench: A Benchmark for API-misuse Detectors. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*.
- [3] Android. 2019. The Activity Lifecycle. <https://developer.android.com/guide/components/activities/activity-lifecycle.html>. (2019). [Online; accessed 4 January 2019].
- [4] Android. 2019. Android API. <https://developer.android.com/guide/index.html>. (2019). [Online; accessed 4 January 2019].
- [5] Android. 2019. Android Studio. <https://developer.android.com/studio/index.html>. (2019). [Online; accessed 21 January 2019].
- [6] M. T. Azim, I. Neamtiu, and L. M. Marvel. 2014. Towards Self-healing Smartphone Software via Automated Patching. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*.
- [7] A. Banerjee, L. Kee Chong, S. Chattopadhyay, and A. Roychoudhury. 2014. Detecting Energy Bugs and Hotspots in Mobile Apps. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*.
- [8] H. Chang, L. Mariani, and M. Pezzè. 2009. In-field Healing of Integration Problems with COTS Components. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
- [9] H. Chang, L. Mariani, and M. Pezzè. 2013. Exception Handlers for Healing Component-based Systems. *ACM Transactions on Software Engineering and Methodologies (TOSEM)* 22, 4 (2013), 30:1–30:40.
- [10] O. Cornejo, D. Briola, D. Micucci, and L. Mariani. 2017. In the field monitoring of software applications. In *Proceedings of the International Conference on Software Engineering (ICSE), NIER track*.
- [11] Y. Dai, Y. Xiang, and G. Zhang. 2009. Self-healing and Hybrid Diagnosis in Cloud Computing. In *Proceedings of the International Conference on Cloud Computing (CloudCom)*.
- [12] G. Denaro, M. Pezzè, and D. Tosi. 2013. Test-and-adapt: An Approach for Improving Service Interchangeability. *ACM Transactions on Software Engineering and Methodologies (TOSEM)* 22, 4 (2013), 28:1–28:43.
- [13] D. Dig, S. Negara, V. Mohindra, and R. Johnson. 2008. ReBA: Refactoring-aware Binary Adaptation of Evolving Libraries. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
- [14] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*.
- [15] S. Elmalaki, L. Wanner, and M. Srivastava. 2015. CAreDroid: Adaptation Framework for Android Context-Aware Applications. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*.
- [16] Y. Falcone, S. Currea, and M. Jaber. 2012. Runtime Verification and Enforcement for Android Applications with RV-Droid. In *Proceedings of the International Conference on Runtime Verification (RV)*.
- [17] Eclipse Foundation. 2019. Acceleo. <https://www.eclipse.org/acceleo/>. (2019). [Online; accessed 21 January 2019].
- [18] Eclipse Foundation. 2019. Eclipse. <http://www.eclipse.org/>. (2019). [Online; accessed 21 January 2019].
- [19] Eclipse Foundation. 2019. Eclipse Modeling Framework. <https://www.eclipse.org/modeling/emf/>. (2019). [Online; accessed 21 January 2019].
- [20] Eclipse Foundation. 2019. Graphical Modeling Project. <http://www.eclipse.org/modeling/gmp/>. (2019). [Online; accessed 21 January 2019].
- [21] JS Foundation. 2019. Appium. <http://appium.io>. (2019). [Online; accessed 4 January 2019].
- [22] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering (TSE)* 45, 1 (2019), 34–67.

- [23] M. E. Joorabchi, A. Mesbah, and P. Kruchten. 2013. Real Challenges in Mobile App Development. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*.
- [24] S. Kelly and J.-P. Tolvanen. 2008. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley.
- [25] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. 2014. Prioritizing the Devices to Test Your App on: A Case Study of Android Game Apps. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
- [26] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.
- [27] D. Kong, L. Cen, and H. Jin. 2015. AUTOREB: Automatically Understanding the Review-to-Behavior Fidelity in Android Applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
- [28] D. Li and W. G. J. Halfond. 2014. An Investigation into Energy-saving Programming Practices for Android Smartphone App Development. In *Proceedings of the International Workshop on Green and Sustainable Software (GREENS)*.
- [29] L. Li, T. F. Bissyandé, D. Oteau, and J. Klein. 2016. DroidRA: Taming Reflection to Support Whole-program Analysis of Android Apps. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.
- [30] Z. Li and Y. Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*.
- [31] J. Ligatti, L. Bauer, and D. Walker. 2005. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security* 4, 1 (2005), 2–16.
- [32] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
- [33] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
- [34] J. Liu, T. Wu, J. Yan, and J. Zhang. 2016. Fixing Resource Leaks in Android Apps with Light-Weight Static Analysis and Low-Overhead Instrumentation. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*.
- [35] J. P. Magalhães and L. M. Silva. 2015. SHÖWA: A Self-Healing Framework for Web-Based Applications. *ACM Transactions on Autonomous and Adaptive Systems* 10, 1 (2015), 4:1–4:28.
- [36] F. Mancinelli and P. Inverardi. 2006. A Resource Model for Adaptable Applications. In *Proceedings of the International Workshop on Self-adaptation and Self-managing Systems (SEAMS)*.
- [37] L. Mariani, F. Pastore, and M. Pezzè. 2011. Dynamic Analysis for Diagnosing Integration Faults. *IEEE Transactions on Software Engineering (TSE)* 37, 4 (2011), 486–508.
- [38] T. McDonnell, B. Ray, and M. Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the International Conference on Software Maintenance (ICSM)*.
- [39] H. Muccini, A. Di Francesco, and P. Esposito. 2012. Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the International Workshop on Automation of Software Test (AST)*.
- [40] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda. 2013. PatchDroid: Scalable Third-party Security Patches for Android Devices. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.
- [41] R. Neisse, G. Steri, D. Geneiatakis, and I. N. Fovino. 2016. A privacy enforcing framework for Android applications. *Computers & Security* 62 (2016), 257–277.
- [42] OMG. 2012. Object Constraint Language. <https://modeling-languages.com/ocl-tutorial/>. (2012). [Online; accessed 21 January 2019].
- [43] O. Riganelli, D. Micucci, and L. Mariani. 2016. Healing Data Loss Problems in Android Apps. In *Proceedings of the International Workshop on Software Faults (IWSF), co-located with the International Symposium on Software Reliability Engineering (ISSRE)*.
- [44] O. Riganelli, D. Micucci, and L. Mariani. 2017. Policy enforcement with proactive libraries. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*.
- [45] O. Riganelli, D. Micucci, and L. Mariani. 2018. Increasing the Reusability of Enforcers with Lifecycle Events. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)*.
- [46] O. Riganelli, D. Micucci, and L. Mariani. 2019. From source code to test cases: A comprehensive benchmark for resource leak detection in Android apps. *Software: Practice and Experience* 49, 3 (2019), 540–548.
- [47] O. Riganelli, D. Micucci, L. Mariani, and Y. Falcone. 2017. Verifying Policy Enforcers. In *Proceedings of the International Conference on Runtime Verification (RV)*.
- [48] O. Riganelli, M. Mobilio, D. Micucci, and L. Mariani. 2019. A Benchmark of Data Loss Bugs for Android Apps. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*.

- [49] R. Rouvoy, M. Beauvois, L. Lozano, J. Lorenzo, and F. Eliassen. 2008. MUSIC: An Autonomous Platform Supporting Self-adaptive Mobile Applications. In *Proceedings of the Workshop on Mobile Middleware: Embracing the Personal Communication Device (MobMid)*.
- [50] Ronny Seiger, Steffen Huber, and Thomas Schlegel. 2018. Toward an execution system for self-healing workflows in cyber-physical systems. *Software & Systems Modeling* 17, 2 (2018), 551–572.
- [51] Z. Shan, T. Azim, and I. Neamtiu. 2016. Finding Resume and Restart Errors in Android Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
- [52] T. Stahl and M. Voelter. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.
- [53] A. Wasylkowski and A. Zeller. 2009. Mining Temporal Specifications from Object Usage. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
- [54] L. Wei, Y. Liu, and S.-C. Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
- [55] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang. 2016. Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps. *IEEE Transactions on Software Engineering (TSE)* 42, 11 (2016), 1054–1076.
- [56] XDA. 2019. Xposed. <http://repo.xposed.info/>. (2019). [Online; accessed 21 January 2019].
- [57] M. Zhang and H. Yin. 2014. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*.

Received January 2018