Increasing the Reusability of Enforcers with Lifecycle Events

Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani(✉)

University of Milano-Bicocca, Viale Sarca 336, 20126 Milan, Italy
{riganelli,micucci,mariani}@disco.unimib.it

Abstract. Runtime enforcement can be effectively used to improve the reliability of software applications. However, it often requires the definition of ad hoc policies and enforcement strategies, which might be expensive to identify and implement. This paper discusses how to exploit lifecycle events to obtain useful enforcement strategies that can be easily reused across applications, thus reducing the cost of adoption of the runtime enforcement technology. The paper finally sketches how this idea can be used to define libraries that can automatically overcome problems related to applications misusing them.

Keywords: Runtime enforcement · Self-healing · Proactive library

1 Introduction

Runtime enforcement techniques are effective solutions for guaranteeing that software applications satisfy certain correctness policies at runtime [17]. When using runtime enforcement, developers are typically in charge of identifying the policies that must be enforced, defining a strategy to enforce them, and finally implementing the software enforcer that applies the strategy.

The enforced policies are often *application-specific*, that is, policies are defined ad hoc for the target application. Working with application-specific policies might be quite expensive. In fact every time a new application is considered, new policies must be identified, and the modelling and implementation activities must be repeated from scratch.

Interestingly policies may also refer to libraries and components that can be reused across applications being themselves eligible for reuse. *Reusable policies* are extremely important because they can alleviate the developers from the burden of identifying both the policies to be enforced and the corresponding enforcement strategies. Developers could simply reuse policies and enforcement

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).

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11247, pp. 51–57, 2018.
https://doi.org/10.1007/978-3-030-03427-6_7

strategies while they reuse libraries, de facto simplifying the application of runtime enforcement techniques.

Unfortunately, the definition of reusable policies and enforcement strategies can be challenging. Since the context of use of a library is not known a priori, a reusable policy and the corresponding enforcement strategy could be defined referring to the operations of the library only. For example, a reusable policy of a library for interacting with the file system may require that a file is opened before any content is written in the file. However, several relevant policies may depend not only on the usage of a library, but also on the behavior of the application that interacts with the library. For instance, a policy that forces an app to close a file before its execution is suspended depends on both the library and the app, and cannot be specified referring to the library only.

There is a popular class of software applications that naturally facilitate both the identification of reusable policies and the definition of enforcement strategies. We call them *life-cycle based applications*. They are applications whose units of composition are modules with an explicitly documented life-cycle model. There is a huge number of life-cycle based applications. For example, Android apps are composed of activities with a known life-cycle model and with callbacks that are invoked when there is a change in the state of the app; similarly Spring applications are composed of components with a known life-cycle model and callbacks. The same applies to many other contexts, such as Web applications, multi-threaded applications, and so on.

Life-cycle based applications have the important advantage of responding to the same life-cycle and implementing the same callbacks, regardless of what a specific application does. Thus policies and enforcement strategies can exploit this information to consider some aspects of the behavior of the application, still remaining reusable. We call these reusable policies *life-cycle based policies* and the corresponding strategies *life-cycle based enforcement strategies*.

We further elaborate the concept of life-cycle based application and policy in Sect. 2. We show how we exploited these concepts to define *proactive libraries*, a class of libraries augmented with reusable enforcement strategies, in Sect. 3. We provide final remarks in Sect. 4.

2 Life-Cycle Based Policies

The life-cycle of a software unit specifies the possible states of the unit and the events that can cause the transition between two states. Units with a non-trivial and well-defined lifecycle are typically executed and managed by a framework that explicitly controls their life, invoking callback methods when there is a state transition. For example, Android activities have callback methods that are invoked when an application is started and suspended. Similarly, Web components have callback methods that are invoked when they are created and destroyed.

These callback methods are pervasively present in life-cycle based applications. For instance, every activity in every Android application implements the

same callback methods. This is an important aspect that eases the definition of both reusable policies and reusable enforcement strategies that can be generally valid for every application of a specific domain. For instance, a policy about an Android library can also refer to callback methods without any loss of generality.

Policies with life-cycle events are particularly relevant. Applications may have to implement non-trivial behaviors in reaction to state transitions [1–3,6,7,9], and this may lead to faulty applications, for instance applications with faulty library interactions [14,21].

Although these policies might be non-trivial to address, they are easy to find in the documentation of libraries and systems and can be the basis for the design of reusable policies. We report below three examples of reusable policies that can be defined for completely different life-cycle based systems.

The `onPause()` method is an Android callback that is automatically executed when a user stops interacting with an activity and is relevant to several correctness policies. For instance, an activity that is paused after acquiring the `Camera` must release it otherwise the camera might be unusable from other activities[1].

In the OSGi Java framework [4], application bundles can be started, stopped, installed, and uninstalled remotely without rebooting. The execution of these operations must obey to specific policies. For example, stopping a bundle requires unregistering every previously registered service [3].

React is a JavaScript library widely used to build encapsulated components that can be composed to create complex Web UIs [5]. Each component has several life-cycle callback methods that can be overridden to execute custom code at particular times in the component's life-cycle. For example, the method `componentWillUnmount()` is invoked immediately before a component is unmounted and destroyed. The library documentation requires applications to implement specific operations when this callback is executed, such as invalidating timers, deleting network requests, or cleaning up subscriptions [6].

Note that all these examples are cases of policies that can be arbitrarily reused across applications since they exploit information about life-cycle events and library APIs. These policies would be impossible to define without exploiting the information about life-cycle events.

In the next section, we show how we exploited this concept to define *proactive libraries*, that is, libraries equipped with life-cycle based enforcement strategies We present proactive libraries in the Android domain because it is the most popular among the application domains described above, and because it has been already used as application domain in related work [11,18,20].

3 Proactive Libraries

Let us refer to the *Plumeria*[2] app, a simple Android app, to illustrate the concept of *proactive library* [19]. *Plumeria* has a fault, that is, one of its activities does not release the camera when it is suspended, as a consequence the camera becomes

[1] https://developer.android.com/guide/topics/media/camera#release-camera.
[2] https://github.com/DonLiangGit/Plumeria.

inaccessible to the other apps of the device. This is a classic resource leak problem that could be avoided by enforcing the policy presented in Sect. 2. In particular, if the camera API is released as a proactive library, this problem would never show up because it would be automatically detected and fixed by the enforcement mechanism embedded in the proactive library.

Proactive libraries are standard libraries augmented with the built-in capability of enforcing reusable policies at runtime.

Figure 1 shows the generation process of proactive libraries. The process distinguishes the development and the runtime phases.

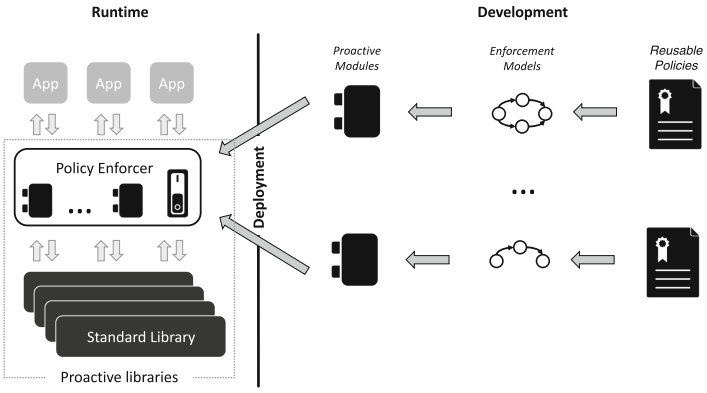

Fig. 1. The generation process of proactive libraries.

At development time, developers start from the identification of reusable correctness policies, that is, natural language statements that specify how the application should use a library according to the status of both the application, detected through the execution of its life-cycle callback methods, and the library, detected through the execution of its API methods. The reusable correctness policy that ensures the correct usage of the camera is: "*An activity that is paused while having the control of the camera must first release the camera.*"

Correctness policies are used to derive enforcement models that define how to react to correctness policies violations. We use edit automata [16] to define the enforcement models because they naturally support the definition of enforcement rules by means of events to be intercepted, inserted and suppressed, and they could be also verified [20]. The definition of an enforcement model does not require any knowledge about the app that uses the API, but it uniquely requires the knowledge of the API and of the Android callback methods, which are the same for any app.

Figure 2 shows a slightly simplified enforcement model that forces the release of the `Camera` when the activity is paused without releasing the `Camera`. The prefix `r` is used to distinguish the calls to the API methods from callbacks. To

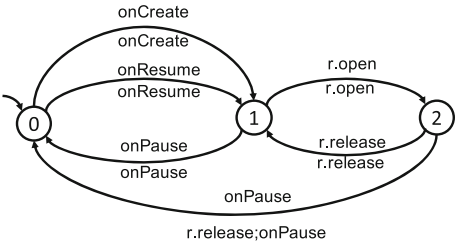

Fig. 2. Simplified enforcement model for the `Camera`.

keep the example real but small, the enforcement model does not include the part that reassigns the `Camera` to the activity once its execution is resumed.

To actually enforce the policy in the target environment, the enforcement models are turned into *proactive software modules* that intercept the execution of life-cycle callback methods and API methods, and produce additional invocations when needed, according to the enforcement model.

Since proactive modules are activated by the invocation of specific 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.

The language and frameworks to implement the proactive modules and the policy enforcer depend on the target environment. In the case of Android, we use the Java Xposed framework [8], which allows to cost-efficiently intercept method invocations and change the behavior of an Android app using run-time hooking and code injection mechanisms.

In our experience, we successfully used proactive libraries to automatically overcome several problems present in Android apps [19].

4 Conclusions

Research on runtime enforcement has already delivered both theoretical [10,12,16,17] and practical results [11,13,15,19]. However, identifying policies, specifying enforcement strategies, and implementing the corresponding enforcers is still a difficult and time consuming task. Reusable policies, as discussed in this paper, can relieve developers from this tedious and error-prone task, facilitating reuse and easing the practical adoption of the runtime enforcement technology.

We plan to extend our work on runtime enforcement in three directions. *Automatic code generation of runtime enforcement mechanisms*: since manually implementing runtime enforcement mechanisms is particularly difficult and expensive, we plan to define a model-driven software development process and the corresponding tool chain to automatically derive enforcer code from the models. *Automatic testing of software enforcers*: To achieve highly reliable and safe enforcing mechanisms, we need techniques specifically defined to validate

the behavior of software enforcers, which have the distinguishing characteristic of being designed to dynamically change the behavior of other software applications, causing hard to predict side effects. *Public repository of software enforcers*: Since life-cycle based enforcement strategies are application-independent, publishing well developed software enforcers in a public repository is important to facilitate the distribution of plug-and-play enforcement strategies that can be easily exploited by developers.

References

1. Apache Felix iPOJO - Lifecycle callbacks. http://tiny.cc/iyvoty
2. Kubernetes - Container Lifecycle Hooks. http://tiny.cc/k9voty
3. OSGi - Life Cycle Layer. http://tiny.cc/k9voty
4. OSGi Alliance - The Dynamic Module System for Java. https://www.osgi.org
5. React - A JavaScript library for building user interfaces. http://tiny.cc/iyvoty
6. React - State and Lifecycle. https://reactjs.org/docs/state-and-lifecycle.html
7. Spring - Customizing the nature of a bean. http://tiny.cc/rs2oty
8. Xposed. http://repo.xposed.info/
9. Android: The Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.html
10. Bielova, N., Massacci, F.: Do you really mean what you actually enforced? Int. J. Inf. Secur. (IS) **10**(4), 239–254 (2011)
11. Falcone, Y., Currea, S., Jaber, M.: Runtime verification and enforcement for android applications with RV-Droid. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 88–95. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_11
12. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? Int. J. Softw. Tools Technol. Transfer **14**(3), 349–382 (2012)
13. Hallé, S., Ettema, T., Bunch, C., Bultan, T.: Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In: Proceedings of the International Conference on Automated Software Engineering (ASE) (2010)
14. Hou, D., Li, L.: Obstacles in using frameworks and APIs: an exploratory study of programmers' newsgroup discussions. In: Proceedings of the International Conference on Program Comprehension (ICPC) (2011)
15. Kumar, A., Ligatti, J., Tu, Y.-C.: Query monitoring and analysis for database privacy - a security automata model approach. In: Wang, J., et al. (eds.) WISE 2015. LNCS, vol. 9419, pp. 458–472. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26187-4_42
16. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. Int. J. Inf. Secur. **4**(1), 2–16 (2005)
17. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. **12**(3), 19:1–19:39 (2009)
18. Riganelli, O., Micucci, D., Mariani, L.: Healing data loss problems in android apps. In: Proceedings of the International Workshop on Software Faults (IWSF), Co-located with ISSRE (2016)
19. Riganelli, O., Micucci, D., Mariani, L.: Policy enforcement with proactive libraries. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) (2017)

20. Riganelli, O., Micucci, D., Mariani, L., Falcone, Y.: Verifying policy enforcers. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 241–258. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_15
21. Wang, W., Godfrey, M.W.: Detecting API usage obstacles: a study of ios and android developer questions. In: Proceedings of the Working Conference on Mining Software Repositories (MSR) (2013)