Detection of Architectural Violations with Archecker

Victor J. Marin
Department of Computer Science
Rochester Institute of Technology, USA
vxm4964@rit.edu

ABSTRACT

As software systems evolve, their architecture tends to diverge from the idealization in the mind of the architects, which can compromise the non-functional requirements of the system. In this project, we have developed Archecker, a tool to help prevent this undesirable effect by detecting unexpected interactions between components. Given a high-level architectural specification, and a mapping from components in such architecture to software entities, our tool computes the architectural violations in the system under analysis so that corrective actions can be taken.

ACM Reference Format:
Victor J. Marin. 2017. Detection of Architectural Violations with Archecker. In *Proceedings of Software Architecture and Product Lines (SWEN-755).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software architecture aims to ensure that quality attribute requirements such as modifiability or security are met by a software system. However, as software systems evolve and increase in size, their actual implementation tends to differ from the designed architecture. Such phenomenon is referred to as architectural erosion [2], which is an undesirable effect given that the satisfiability of the non-functional requirements of the system can be compromised [1].

Architecture conformance aims to tackle this issue by checking that the implementation of a system is in accordance with its architectural design. In this context, reflexion models [4] have proved to be one of the most successful approaches. They comprise an expected high-level specification of the system's architecture, a source model representing dependencies between software artifacts, and a mapping between elements in the high-level specification and the source model. With this information, they compute a reflexion model which provides a comparison between the expected architecture and the real implementation. Using the reflexion model, the architect can detect architectural violations and apply corrective actions to fight architecture erosion.

This project sought to develop a tool for checking architecture compliance. To that end, we have developed Archecker, a tool capable of detecting architectural violations and generating reflexion models for Java projects. Archecker is publicly available at https://github.com/victorjmarin/Archecker/.

SWEN-755, December 2017, Rochester, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND

The Software Reflexion Model Technique, as originally proposed by Murphy et al. [4], comprises the following steps:

(1) The architect designs an intended architecture defining components and their expected interactions.
(2) The architect extracts structural information from the artifacts in the software system to create a source model.
(3) The architect specifies a mapping between elements in the source model and the architecture.
(4) A software reflexion graph which provides a comparison between the intended architecture and the source model is computed. Nodes in such graph represent components in the intended architecture. Edges represent the following relationships:
   - A solid edge indicates convergence, meaning that the interaction was expected.
   - A dashed edge indicates divergence, meaning that the interaction was unexpected.
   - A dotted edge indicates an absence, meaning that no instances of an expected interaction occur in the source model.
(5) Using the reflexion model, the architect can detect architectural violations and apply corrective measures to fight architecture erosion.

3 THE ARCHECKER TOOL

This section describes how Archecker detects architectural violations and builds a reflexion model from a software project.

3.1 Required Specifications

Archecker requires three elements to perform the architecture conformance checking. In the first place, it takes the Java project to be analyzed as a .jar file. Secondly, it needs an *architectural specification* stating the expected interactions between components. Lastly, it requires a *mapping specification* from components in the architectural specification to actual software entities in the project.

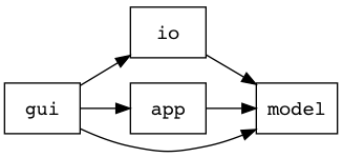

Figure 1: Conceptual architectural specification as a graph. Nodes represent components, and edges expected interactions amongst them.

Conceptually, an architectural specification is a graph in which nodes represent components and edges indicate expected interactions amongst such components. Figure 1 shows an example of an architectural specification. We can distinguish four components, namely the `gui`, `io`, `app` and `model` components. In addition, we expect the `gui` component to interact with all the three remaining components; the `io` and `app` components to interact with the `model` component, and the `model` component to interact with none of the others. In practice, architectural specifications are provided as a JSON file consisting of an array of objects with two attributes each. The first attribute is the `component` attribute, which is a string referring to the name of the architectural component defined by that object. The second attribute is the `should-call` attribute, which is an array of strings enumerating other components in the specification that the component under definition is expected to interact with. Figure 2 illustrates an equivalent JSON representation of the architectural specification in Figure 1 that can be interpreted by Archecker.

```
[
   {
      "component":"gui",
      "should-call":["io", "model", "app"]
   },
   {
      "component":"io",
      "should-call":["model"]
   },
   {
      "component":"app",
      "should-call":["model"]
   },
   {
      "component":"model",
      "should-call":[]
   }
]
```

Figure 2: Architectural specification in JSON format.

A mapping specification is a function from components in the architectural specification to software entities in the project under analysis. Similar to architectural specifications, mapping specifications are also provided as JSON files. They consist of an array of objects with two attributes: 1) The `component` attribute is an string denoting the name of a component defined in the architectural specification, and 2) The `entities` attribute is a list of software entities that map to the component in the first attribute. Software entities are referenced using their fully qualified name, and can either be packages or classes in the project implementing behavior associated with the component they are mapped to. Figure 3 illustrates a mapping from components defined in Figure 2 to entities in the associated project under analysis. For example, the `gui` component is realized by software artifacts inside the package `net.sf.jmoney.gui`, and the package `net.sf.jmoney.io` is mapped to the `io` component. Even though not shown in the example, a component can have multiple software entities mapped to it.

3.2 Architecture Conformance Checking

Given the architectural and mapping specifications for the project under analysis as JSON files, Archecker first builds an in-memory model and performs some integrity checks. The checks involve

```
[
   {
      "component":"gui",
      "entities":["net.sf.jmoney.gui"]
   },
   {
      "component":"io",
      "entities":["net.sf.jmoney.io"]
   },
   {
      "component":"app",
      "entities":["net.sf.jmoney"]
   },
   {
      "component":"model",
      "entities":["net.sf.jmoney.model"]
   }
]
```

Figure 3: Mapping specification in JSON format.

making sure that all the components referenced in the mapping specification are previously defined in the architectural specification, and that all defined components have mapped entities. Once the architectural model is built, Archecker proceeds to build a call graph which will be used as the source model. To that end, it scans the files in the provided .jar project processing only those with the .class extension, i.e., the compiled bytecode corresponding to the source files. Recall that entities could be either classes or packages. For every .class file, we first try to use its package to find a matching entity in the mapping specification. If no matching entity is found based on the package, we attempt to use the class name. All calls found within the .class file will be associated with the matched entity. Then, the tool uses the visitor pattern to go through bytecode instructions corresponding to method invocations, namely `invokedynamic`, `invokeinterface`, `invokespecial`, `invokestatic` and `invokevirtual` instructions. By searching for method invocations in this way, we are guaranteed to find all method invocations and associate them with their correct entity. For each call, we create an object representation that stores the line where it happened, the caller and the callee, as well as the invoker and invoked methods. Calls to methods inside packages or classes not in the mapping specification are discarded since they will not play any role in the architecture conformance checking. The process of finding all relevant calls in .class files is parallelized to achieve a higher throughput and make it scalable to projects with a large number of classes.

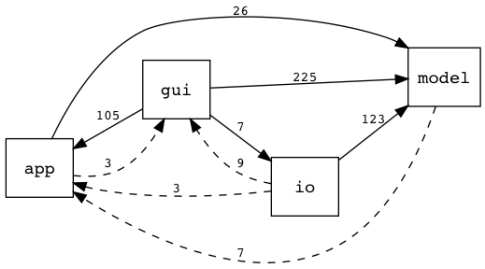

Figure 4: Reflexion graph.

After going through the .class files in the project, the in-memory model will have all the calls performed by every entity. Therefore,

```
(37) net.sf.jmoney.Start.<init>() ->
        net.sf.jmoney.gui.MainFrame.<init>()
(38) net.sf.jmoney.Start.<init>() ->
        net.sf.jmoney.gui.MainFrame.show()
(39) net.sf.jmoney.Start.<init>() ->
        net.sf.jmoney.gui.MainFrame.initProperties()
```

Figure 5: Excerpt of calls incurring an architectural violation.

we are in a position where we can check for architectural violations, i.e., calls between components that are not expected per the architectural specification. The algorithm is linear in the number of calls, checking if the component that the caller entity maps to is allowed to interact with the component that the callee entity maps to for every call. If they are allowed to interact then the call constitutes a convergence, and it is a divergence otherwise. At the end, absences are trivially computed. The algorithm outputs a reflexion graph and a text file containing all the architectural violations. Figure 4 illustrates a reflexion graph for a given project using the specifications in the previous figures. We can notice that there are 225 convergences from `gui` to `model` or 7 divergences from `model` to `app`. Figure 5 shows excerpt of calls incurring an architectural violation displaying the line, the caller method and the callee method involved in the violation.

4 EVALUATION

We evaluated Archeckr in two different scenarios. The first scenario consisted of a synthetic project with over 500K calls spread between five classes. In such context, our tool was capable of building a reflexion model and reporting all 400K calls that constituted an architectural violation in under four seconds using commodity hardware.

In the second scenario, we tried to reproduce the results reported in [3] regarding the private finance tracker software JMoney (version 0.4.4). Figure 4 shows the reflexion graph computed by Archecker using the specifications in Figures 2 and 3. We found some discrepancies with respect to the reflexion graph discussed in such paper. For example, from `gui` to `io` they report 10 convergences while we report only 7. We performed a thorough manual inspection of calls from one component to the other and could only locate 7 calls indeed. Therefore, our hypothesis is that they are not only using calls to build the reflexion graph, but some other kind of dependencies too. However, we were not able to test such hypothesis during our investigations. In the original paper introducing reflexion models, only call dependencies are considered [4].

We tried looking for additional case studies, such as [1, 5], but they involved projects not publicly-available.

5 CONCLUSIONS

In this project we explored the use of software reflexion models as a means to check architecture compliance and prevent architecture erosion. To that end, we developed Archecker, a tool capable of detecting architectural violations and generating reflexion models for Java projects. Future work includes performing evaluation on more case studies, and making the tool configurable to support several source model configurations beyond method invocations.

REFERENCES

[1] Jim Buckley, Nour Ali, Michael English, Jacek Rosik, and Sebastian Herold. 2015. Real-Time Reflexion Modelling in architecture reconciliation: A multi case study. *Information & Software Technology* 61 (2015), 107–123. https://doi.org/10.1016/j.infsof.2015.01.011

[2] Lakshitha de Silva and Dharini Balasubramaniam. 2012. Controlling software architecture erosion: A survey. *Journal of Systems and Software* 85, 1 (2012), 132–151. https://doi.org/10.1016/j.jss.2011.07.036

[3] Sebastian Herold, Michael English, Jim Buckley, Steve Counsell, and Mel Ó Cinnéide. 2015. Detection of Violation Causes in Reflexion Models. In *SANER*. 565–569. https://doi.org/10.1109/SANER.2015.7081878

[4] Gail C. Murphy and David Notkin. 1997. Reengineering with Reflection Models: A Case Study. *IEEE Computer* 30, 8 (1997), 29–36. https://doi.org/10.1109/2.607045

[5] Jacek Rosik, Andrew Le Gear, Jim Buckley, and Muhammad Ali Babar. 2008. An industrial case study of architecture conformance. In *ESEM*. 80–89. https://doi.org/10.1145/1414004.1414019