# The Build Dependency Perspective of Android's Concrete Architecture

Wei Hu, Dan Han, Abram Hindle, Kenny Wong
*University of Alberta*
*Department of Computing Science*
*Edmonton, Canada*
{*whu4, dhan3, abram.hindle*}*@ualberta.ca, kenw@cs.ualberta.ca*

***Abstract*—Android is an operating system designed specifically for mobile devices. It has a layered architecture. In this paper, we extract Android's concrete layered architecture by analyzing the build dependency relation between Android sub-projects and use it to validate the proposed conceptual architecture. Our experiment shows that Android's concrete architecture conforms to the conceptual architecture. Apart from that, we also show the extracted architecture can help developers and users better understand the Android system and further demonstrate its potential benefits in studying the impact of changes.**

***Keywords*-Dependency; Android; Architecture**

## I. Introduction

Android is an operating system designed specifically for mobile devices. It has a layered architecture. As shown in Figure 1, Android's conceptual architecture consists of four layers: Applications, Application Framework, Libraries and Android Runtime and Linux Kernel. Linux Kernel layer provides core system service like memory management and driver models; Android Runtime layer is built on the top of the Linux Kernel layer and hosts Java Virtual Machine and Java core programming library; Libraries layer provides a set of C/C++ libraries used by other system components; Application Framework layer offers a development platform for developers and manages upper layer applications; Applications layer consists of Java applications that are built on the top of Application Framework layer [1].

In a layered architecture, the build and execution of upper layer components must depend on lower layer components [2], [3]. Hassan *et al.* [2] and Grosskutth *et al.* [3] leverage such dependency relations to validate and refine the proposed conceptual architecture. They use the dependency relations to discover concrete software architecture and subsequently compare the concrete architecture with the conceptual architecture. Wermelinger *et al.* [4] also build Eclipse Plugins' concrete structure using dependency relations.

This paper seeks to validate the Android's conceptual architecture using the existing techniques in [2], [3]. We infer Android's concrete architecture by studying the dependency relations between Android sub-projects and subsequently compare the concrete architecture with the conceptual architecture. Apart from the validation, we also demonstrate the concrete architecture's potential benefits in helping developers and users to better understand the Android system and in studying the impact of change, e.g. how changes in a single sub-project can affect other sub-projects.

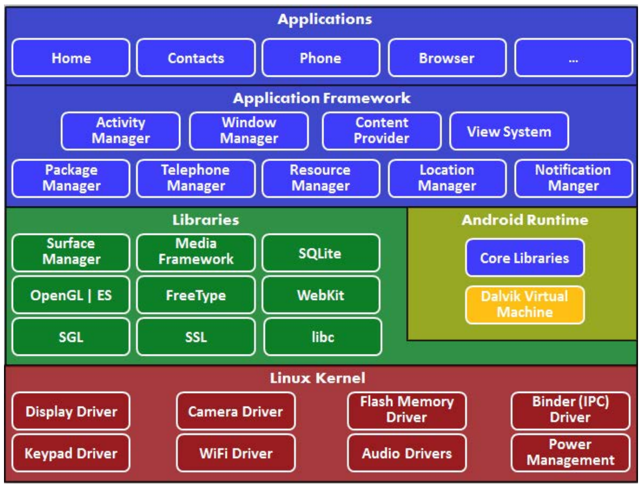

Figure 1: Android's Conceptual Architecture. [1]
This figure is reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

We seek to answer the following questions:

1) What does the concrete architecture look like? Does it follow the conceptual architecture?
2) What's the benefit of the extracted concrete architecture?

The remainder of this paper is organized as follows: Section II describes how we extract the Android's concrete architecture from build dependency relations; Section III presents our experiment results with a dependency DAG (directed acyclic graph), compares the Android's concrete architecture against the conceptual architecture and demonstrates the extracted concrete architecture's potential benefits; Section IV ends this paper with conclusion and future work.

## II. Extract Concrete Android Architecture

German *et al.* [5] originally defines the concept of inter-dependencies as a package as "the set of packages that are required to build and execute the package, but are not distributed with the original application".

Take the case of Android system, it has a set of sub-projects that are combined in a fashion that each sub-project has a separate folder and also a separate build system but its build and execution may require the build of other sub-projects (like the execution of applications written in Java requires the build of Android Java Virtual Machine). In this paper, we denote dependencies of a sub-project of the set of sub-projects that are required to build that sub-project.

To extract the Android's concrete architecture, we first identify the dependency relations, relate sub-projects to their dependencies and then visualize the Android layered architecture with a dependency-DAG (directed acyclic graph). The extracted concrete architecture follows the development view in [6] since it focuses on the software modules in the development environment. As shown in Figure 2, the process for extracting the Android's concrete architecture consists of the following steps:

### A. *Build from Source Code to Generate the Build Trace*

As Android-4.0.1 employs "make" [7] to build the sub-projects, we compile the non-kernel source code using "make" in its debugging mode to get the build trace. The exact debugging flags we use is "`-w --debug=v --debug=m -p`".

Since the Android kernel has its own build system and the build of kernel does not interact with the non-kernel source code, we only compile and mine the Android-4.0.1 non-kernel source code in this paper.

### B. *Identify the File Level Dependency Relations*

We use MAKAO [8] (a reverse-engineering framework for build systems) to automatically analyze the build trace of Android to discover the dependency relations between files. After gaining the knowledge of file level dependency relations and files mapped to sub-projects, we subsequently identify the dependency relations between sub-projects.

### C. *Infer the Dependency Relations between Sub-projects*

We infer the dependency relations between sub-projects from the known file level dependency relations. To better explain the identify process, we use Figure 3 as an example. In Figure 3, the concrete directed lines represent build dependent relations between objects and the dashed directed lines stand for dependency relations between sub-projects. For example, the red line labeled with "depend" means the the build of object *baz* depends on the object *foo.o*.

Theoretically, sub-project *foo* can be identified as sub-project *bar*'s dependency if the build of sub-project *bar* depends on objects in the folder of sub-project *foo*. However, in Android, instead of keeping the intermediate object files locally in its own folder, each sub-project outsources the generated binaries and libraries to a particular public folder

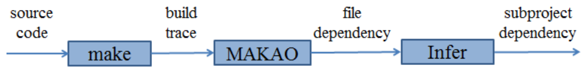

Figure 2: The process for extracting the Android's concrete architecture.

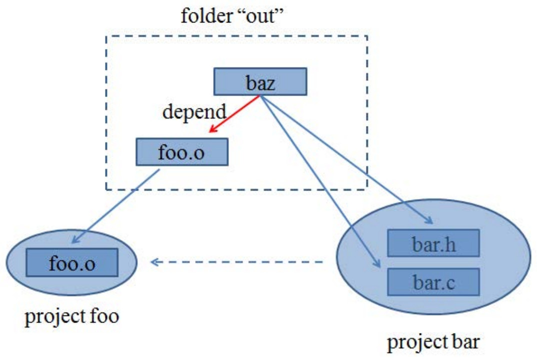

Figure 3: Infer the dependency relations between sub-projects.

called *"out"*. So in the case of Figure 3, instead of depending on the objects in sub-project *foo*'s folder, the build of sub-project *bar* requires the build of object *foo.o* which was originally generated by sub-project *foo* but is now being placed in folder *"out"*.

To handle the offloaded intermediate object files, we adopt a slightly-modified identification algorithm. At a high level, We mark sub-project *foo* as sub-project *bar*'s dependency if there exists an object *baz* in folder *"out"* such that the production of *baz* depends on libraries and headers (.o, .so, .h and .jar files) from sub-project *foo* and source files (.java, .c and .cpp files) from sub-project *bar*. In this case, sub-project *foo* is a dependency of sub-project *bar*.

## III. Experiment and Discussions

### A. *Experiment*

Our algorithm extracts the dependency relations among 90 sub-projects out of 240 sub-projects. The generated dependency DAG (directed acyclic graph) is processed by GUESS [9] (an exploratory data analysis and visualization tool for graphs and networks) and is shown in Figure 4.

In Figure 4, each node represents one sub-project, each directed edge indicates one sub-project connects to one of its dependencies. We manually classify the 90 sub-projects into the class of applications, application framework, libraries, android runtime and linux kernel by their functionality. We handle the classification by checking both the project name and documentation. For example, we classify the sub-project "framework/base" into the class of framework because this sub-project is prefixed with "framework" and is documented to provide core framework libraries.

### B. *Discussions*

**1) What does the Android's Concrete Layered Architecture Look Like?**

- Applications Layer: *Most of the applications depend on sub-project "frameworkBase" at application framework layer and sub-project "libCore" at runtime layer; Part of the applications depend on sub-projects at libraries layer and prebuilt kernel binary.*
  This observation verifies that applications are built on the top of the application framework layer, runtime layer, libraries layer and linux kernel layer.
- Application Framework Layer: *Most of the sub-projects at application framework layer depend on sub-project "libCore" at runtime layer, part of the sub-projects at application framework layer depend on prebuilt kernel binary and one application framework layer sub-project "mock" depends on sub-project "javassist" at libraries layer*
  This observation verifies that application framework layer is built on the top of the libraries layer, runtime layer and linux kernel layer.
- Libraries Layer: *Part of the sub-projects at libraries layer depend on sub-project "libCore" at the runtime layer and part of the sub-projects at libraries layer depend on prebuilt kernel binary*
  This observation verifies that the libraries layer is built on the top of the linux kernel layer
- Android Runtime Layer: *Sub-project "Dalvik" depends on prebuilt kernel binary*
  This observations verifies that the runtime layer is built on the top of the linux kernel layer
- No Exception: *There exists no lower layer sub-project whose build depends on an upper layer sub-project and there are no circular dependencies*
  In terms of "upper" and "lower", we mean the layer defined in Android's conceptual architecture

The extracted Android's concrete architecture, or to say the dependency DAG, actually validates the conceptual architecture in [1]. All the upper layer is built on the top of the lower layer and there exists no lower layer sub-project whose build depends on an upper layer sub-project. Furthermore, there are no circular dependencies.

**2) What are the Potential Benefits of the Extracted Android Concrete Architecture?**

- *The Android's concrete architecture provides more information for developers and users than the conceptual architecture does.*

Although the Android's conceptual architecture is validated by the extracted concrete architecture, it provides only general layer-level dependency relations. In other words, it does not specify any particular sub-project's dependencies. Furthermore, there are only 16 non-kernel sub-projects involved in the conceptual architecture in Figure 1. As a comparison, the extracted concrete architecture depicts a clearer view in the sense that it indicates exactly which sub-project depends on which sub-project and hence can help developers and users better understand the Android system.

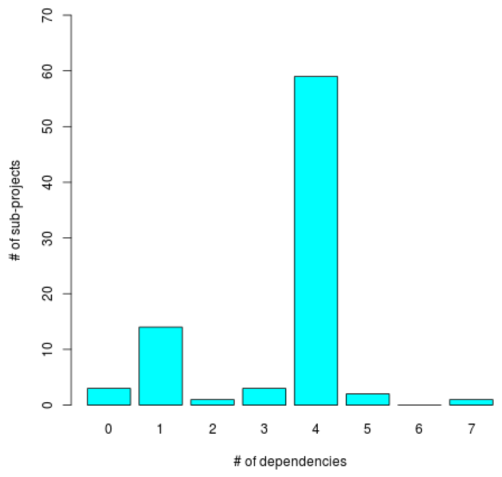

Figure 5: Bar Chart of the number of sub-projects with a given number of build dependencies.

- *The dependency relation is valuable for understanding the impact of changes.*

Simply put, changes in a dependency may affect sub-projects that depend upon it, consequently the change in the bottom of the dependency DAG may traverse upwards to affect the top sub-project. So the dependency DAG provides meaningful information for understanding the impact of changes.

In order to measure the impact of change, we count how many dependencies (directly and transitively dependent) each sub-project has by performing a depth-first search on the dependency DAG. The heavily skewed distribution of dependencies is depicted in Figure 5. It implies that most of the sub-projects have four dependencies.

Table I: Top four most common dependencies

| Project name | number of sub-projects depend upon it |
|---|---|
| platform/frameworks/ex | 65 |
| platform/frameworks/base | 65 |
| platform/libcore | 71 |
| platform/prebuilt | 76 |

Correspondingly, to explain why the distribution of dependencies is skewed, we find out the top four most common dependencies (directly and transitively dependent) in the process of depth-first search and show it in Table I. Sub-projects "platform/frameworks/ex" and "platform/frameworks/base" provide basic application framework service; "platform/libcore" is the Android Java core programming language library and "platform/prebuilt" is the prebuilt kernel binary. Since sub-project "platform/frameworks/base" contains core framework libraries,

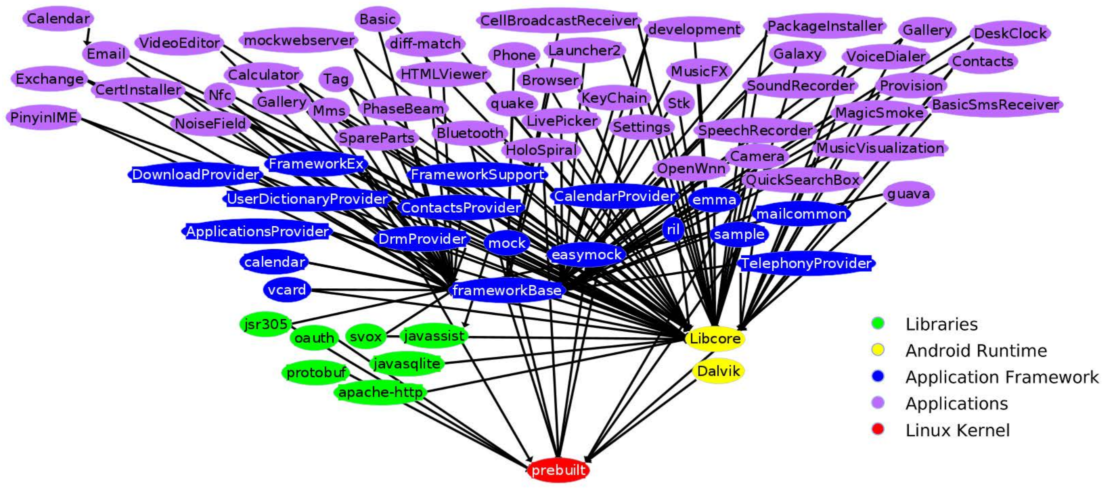

Figure 4: Sub-projects dependency DAG (directed acyclic graph).

it is depend upon by all the applications. Consequently, as sub-project “platform/frameworks/base” depends on sub-projects “platform/frameworks/ex” and ”platform/prebuilt”, it makes all the applications depend on sub-projects “platform/frameworks/ex” and ”platform/prebuilt” transitively. Moreover, as all the Android applications and most of the framework sub-projects are written in Java, sub-project “platform/libcore” is depend upon by all the applications and most of the framework layer sub-projects. As a conclusion, these four sub-projects actually provide the basic service for other Android components and thus are depend upon by the most majority of Android sub-projects. Consequently, a single bug or change in any of these four sub-projects may affect all the other Android sub-projects.

## IV. Conclusion and Future work

In this paper, we have mined the Android concrete architecture. From the extracted concrete layered architecture, we validate that the Android’s concrete architecture conforms to the conceptual architecture. Furthermore, we showed that the concrete architecture provides much more valuable information than the conceptual architecture offers and demonstrated the extracted concrete architecture’s potential benefits in studying the impact of changes.

In the future, we plan to extend our work from project level dependency relation mining down to file level dependence relation mining. Since in the building, the source files propagate information from the bottom dependencies to the top dependent objects, we can study the information aggregation and distribution during the build process in a social-network perspective.

## Acknowledgment

Thanks to Eleni Stroulia and anonymous reviewers for their valuable comments.

## References

[1] Android developer guide. [Online]. Available: http://developer.android.com/guide/basics/what-is-android.html

[2] A. Hassan and R. Holt, “A reference architecture for web servers,” in *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, 2000, pp. 150 –159.

[3] A. Grosskurth and M. Godfrey, “A reference architecture for web browsers,” in *Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on*, sept. 2005, pp. 661 – 664.

[4] M. Wermelinger and Y. Yu, “Analyzing the evolution of eclipse plugins,” *Data Processing*, p. 133, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1370750.1370783

[5] D. German, J. Gonzalez-Barahona, and G. Robles, “A model to understand the building and running inter-dependencies of software,” in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, oct. 2007, pp. 140 –149.

[6] P. Kruchten, “The 4+1 view model of architecture,” *IEEE Softw.*, vol. 12, pp. 42–50, November 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=624610.625529

[7] F. B. Laboratories and S. I. Feldman, “Make — a program for maintaining computer programs,” vol. 9, pp. 255–265, 1979.

[8] B. Adams, K. De Schutter, H. Tromp, and W. D. Meuter, “Makao (demo),” in *Proceedings of the 23rd International Conference on Software Maintenance (ICSM)*, L. Tahvildari and G. Canfora, Eds. Paris, France: IEEE Computer Society, October 2007, pp. 517–518.

[9] E. Adar and M. Kim, “Softguess: Visualization and exploration of code clones in context,” in *In the proceedings of the 29th International Conference on Software Engineering (ICSE07), Tool Demo*, 2007, pp. 762–766.