Angepasstes Android

Ein Tauchgang in die Untiefen der Android-Anpassungsarchitektur

Martin Becker
Fraunhofer Institut für Experimentelles Software Engineering (IESE)

Das Android Betriebssystem wird seit Jahren in einer Vielzahl von eingebetteten Systemen eingesetzt. Dabei kann es hochgradig und weitreichend an die spezifischen Einsatzkontexte angepasst werden. Dies gelingt durch ein ausgefeiltes Zusammenspiel von unterschiedlichen Mechanismen auf verschiedenen Systemebenen. Diese Konfigurations- und Anpassungsarchitektur ist ein sehr guter Ideengeber, wie man in der eigenen Systemlandschaft notwendige Anpassungen effizient umsetzen und beherrschen kann. Leider ist es nicht ganz einfach, sich einen entsprechenden Überblick zu verschaffen. Der Beitrag beleuchtet daher, welche Arten von Anpassungen in Android unterstützt werden und wie diese in der Android-Architektur umgesetzt sind.

Motivation

Der Bedarf an Softwarelösungen, die passgenau auf den Kundenbedarf zugeschnitten sind, steigt ungebremst. Im Entwicklungsalltag stellt sich hier oftmals die Frage, wie man die notwendigen Anpassungen geschickt unterstützen kann. In früheren Beiträgen haben wir hierzu typische Mechanismen und Architekturen betrachtet. In diesem Beitrag beleuchten wir, wie die notwendigen Anpassungen im Open-Source-Betriebssystem Android unterstützt werden.

Android wird seit einigen Jahren in einer Vielzahl von eingebetteten Systemen sehr erfolgreich eingesetzt. Neben der mobilen Unterhaltungselektronik finden sich auch Anwendungen im Automobilbereich und der Gesundheitsversorgung [1]. Die Codebasis von Android ist sehr groß und umfasst einen erheblichen Anteil von anderen Open-Source-Projekten. Der breite Einsatz von Android lässt vermuten, dass es hochgradig an die spezifischen Einsatzkontexte angepasst werden kann. Neben einer Vielzahl von Hardwareplattformen werden auch weitreichende Anpassungen des Funktionsumfangs unterstützt. Dies gelingt durch ein ausgefeiltes Zusammenspiel von unterschiedlichen Mechanismen auf verschiedenen Ebenen. Diese Konfigurations- und Anpassungsarchitektur ist ein guter Ideengeber, wie man in der eigenen Systemlandschaft notwendige Anpassungen effizient umsetzen und beherrschen kann. Leider ist es nicht ganz so einfach, sich einen entsprechenden Überblick zu verschaffen. Um hier Abhilfe zu leisten, haben wir das Android-Buildsystem tiefergehend analysiert. Dieser Beitrag stellt die Analyseergebnisse aus [2][3] kurz vor. Er beleuchtet dabei, welche Arten von Anpassungen in Android unterstützt werden und wie diese in der Android-Anpassungsarchitektur umgesetzt sind.

Android Code-Repository

Unserer Analyse liegt das Code-Repository von Android 6 [4] zugrunde. Obwohl Android eine Vielzahl von anderen Open-Source-Projekten verwendet, haben wir das

komplette Code-Repository analysiert. Der Repository-Inhalt ist wie in Abb. 1 dargestellt sehr groß und im Hinblick auf die verwendeten Dateitypen auch recht heterogen. Offenkundig kommen mit Java, C, C++, Python, Go und Javascript eine Reihe von Programmiersprachen zum Einsatz, wobei der Großteil von Android in C/C++ programmiert ist. Der Buildprozess wird über 4475 Makefiles gesteuert – eine durchaus stattliche Anzahl, die sich nicht einfach überblicken lässt.

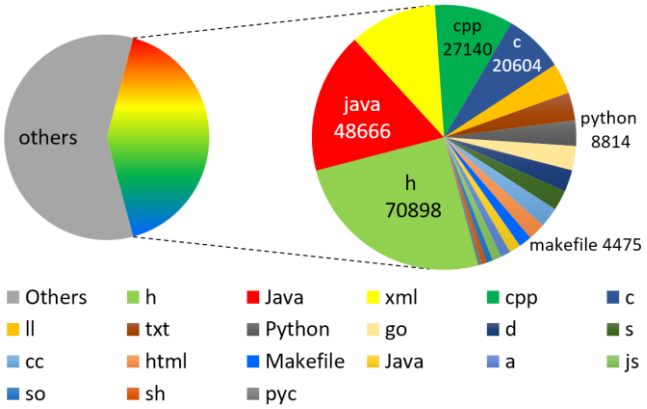

Abb. 1: Übersicht über das Android Code-Repository

Der Code ist grob über drei Ebenen verteilt (siehe Abb. 2):

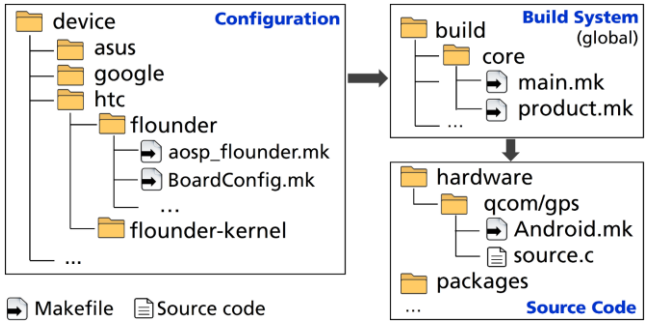

Abb. 2: 3-Ebenen-Architektur des Android-Source-Repositories

Die Android Configuration-Ebene umfasst Produkt- und Boardkonfigurationen. Letztere befinden sich hauptsächlich im Ordner `Device`. Einige generische Produktkonfigurationen finden sich in `/build/target/product`. Die entsprechenden Configuration-Items werden in Makefiles definiert, welche sich typischerweise in `device/<vendor_name>/<device_name>` befinden. Dadurch lassen sich diese im Buildsystem und im Quellcode verwenden. Produktkonfigurationen umfassten generelle Produktinformationen, wie Produktname und Hersteller, sowie eine Liste von Modulen, die für ein Gerät benötigt werden. Boardkonfigurationen werden immer in einem Makefile namens `BoardConfig.mk` festgelegt, in denen eine

Vielzahl von Parametern rund um Peripheriegeräte auf dem Board gesetzt werden. Diese Parameter werden in den Codemodulen verwendet.

Das Android Buildsystem besteht aus zwei Teilen: i) globalen Builddateien, die den übergeordneten Buildprozess festlegen und ii) lokalen Builddateien, die einzelne Module bauen. Wie in Abb. 2 dargestellt, befinden sich die globalen Builddateien im `build` Ordner. Das Makefile `main.mk` stellt den Einstiegspunkt in den Buildprozess dar. Das Makefile `product.mk` definiert Hilfsfunktionen, z.B. um Produktabhängigkeiten aufzulösen. In jedem Codemodul existiert ein Makefile namens `Android.mk`, welches das Modul baut. Makefiles des Buildsystems können Konfigurationsvariablen, die in den Konfigurationsdateien definiert wurden, verwenden.

Android Quellcode ist in Module strukturiert, die sich in getrennten Ordnern befinden. Die Ordner sind dabei hierarchisch geschachtelt. Einige davon gehören zur Android Basisinfrastruktur (z.B. der Linux Kern), andere gehören zur Applikationsunterstützung (z.B. Phone). Einige Module sind Hersteller-spezifisch und befinden sich deshalb im `device` Ordner.

Anpassungen auf der Konfigurationsebene

Konfigurationen werden über Makefile-Variablen definiert, die Software- und Hardware-Einstellungen vornehmen. Die entsprechenden Makefiles befinden sich im `device` Ordner. Insgesamt haben wir mit einem selbstgebauten Parser 206 Konfigurationsvariablen gefunden. Die meisten Variablen folgen der Namenskonvention `PRODUCT_*` oder `BOARD_*`. Unsere Analyse zeigte, dass ein Großteil der Produktvariablen lediglich zu Beschreibungszwecken verwendet wird. Im Gegensatz dazu, werden die Boardvariablen zur Steuerung des Buildprozesses verwendet. Dabei wird eine recht kleine Anzahl von Variablen häufig referenziert, während die Mehrzahl der Variablen nur an wenigen Stellen verwendet wird. Hier hinterlässt das Buildsystem einen recht aufgeräumten Eindruck. Eine Übersicht hierzu findet sich in Tab. 1. Dabei werden Referenzen im globalen `build` Ordner, in Hersteller-spezifischen Modulen im `device` Ordner sowie sonstigen Modulen unterschieden. Zu den am meistverwendeten Konfigurationsvariablen zählen `TARGET_ARCH` und `TARGET_BOARD_PLATFORM`, was letztlich zu erwarten war.

# Config References	1	2~3	4~50	51~246	Sum
# Configs in Global	19	23	41	3	**86**
# Configs in Device	6	6	4	0	**16**
# Configs in Others	38	28	25	3	**94**

Tab. 1: Referenzen auf Konfigurationsvariablen im Buildsystem

Abb. 3 zeigt eine typische Verwendung der Konfigurationsvariablen in den Makefiles. Hier werden Abhängigkeiten rund um dex-preoptimization[1] umgesetzt. In Abhängigkeit vom Wert von übergeordneten Variablen wird der Wert von anderen Variablen gesetzt.

```
# Enable dex-preoptimization to speed up boot
ifeq ($(HOST_OS), linux)
  ifeq ($(TARGET_BUILD_VARIANT), user)
    ifeq ($(WITH_DEXPREOPT),)
      WITH_DEXPREOPT := true
    endif
  endif
endif
```

Abb. 3: Abhängigkeiten zwischen Konfigurationsvariablen

Zwischen den Produktkonfigurationen wird eine Art von Vererbung unterstützt. Dabei kann ein Makefile von mehreren Parent-Makefiles Einstellungen erben. Dies wird durch die Hilfsfunktion `inherit-product` in `product.mk` ermöglicht. Hierdurch lassen sich gemeinsame Einstellungen einfacher wiederverwenden. Dabei sind Einstellungen in den übergeordneten Makefiles als Default-Werte zu sehen, die in spezifischeren Produktkonfigurationen überschrieben werden können. Der Vererbungsgraph zwischen den Produktkonfigurationsdateien ist in Abb. 5 dargestellt. Er lässt sich mit `make product-graph` erstellen. Wie man sieht, kommt dabei auch an mehreren Stellen eine Mehrfachvererbung zum Einsatz. Hier muss darauf geachtet werden, dass die Belegungen richtig überlagert werden.

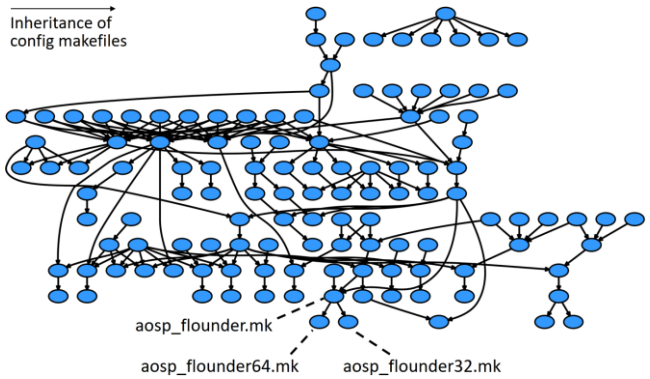

Abb. 4: Vererbung von Produktkonfigurationen

Anpassungen auf der Buildsystem-Ebene

Im Gegensatz zu anderen Make-basierten Buildsystemen werden in Android Makefiles nicht rekursiv aufgerufen. Globale Makefiles tragen zunächst alle `Android.mk` Dateien innerhalb der Code-Basis zusammen, inkludieren deren Inhalt und führen dann

[1] Optimierung von Libraries und Apps, um einen schnelleren Start von Android zu ermöglichen

die Make-Anweisungen aus. Wieso wird das so gemacht? Es ermöglicht eine einfache Analyse und Optimierung des Buildprozesses. Doch dazu später mehr.

In den Modul-Makefiles werden dann unter anderem die Dateien ausgewählt, aus denen das Modul gebaut wird. Abb. 6 zeigt hierfür ein typisches Beispiel. Es handelt sich dabei um eine Art von Polymorphismus auf Dateiebene. Dateien, die optional oder alternativ zum Bauen verwendet werden, lassen sich dadurch einfach ermitteln. Man verzichtet wohl gezielt auf die Verwendung des Präprozessors (#ifdefs) zugunsten von Module-Replacement, wodurch sich die Verständlichkeit der Quellcode Dateien erhöht.

```
Ifeq ($(TARGET_ARCH), arm)
    LOCAL_SRC_FILES += \
        src/asm/pvmp3_polyphase_filter_window_gcc.s \
        src/asm/pvmp3_mdct_18_gcc.s \
        src/asm/pvmp3_dct_9_gcc.s
else
    LOCAL_SRC_FILES += \
        src/pvmp3_polyphase_filter_window.cpp \
        src/pvmp3_mdct_18.cpp \
        src/pvmp3_dct_9.cpp
endif
```

Abb. 5: Auswahl von Dateien, die beim Bauen verwendet werden

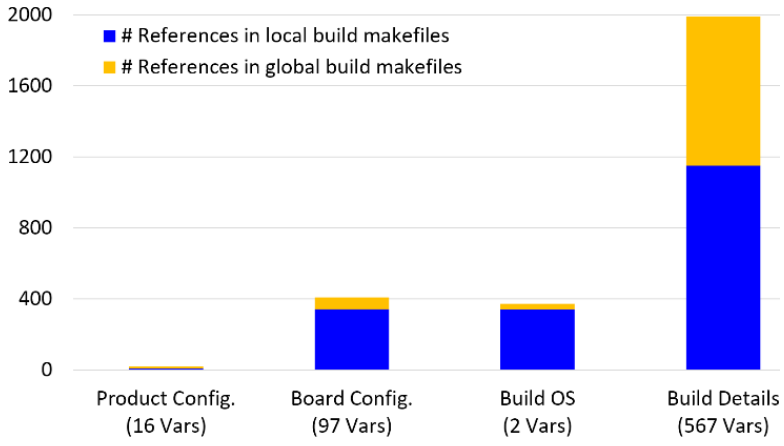

Abb. 6: Verwendung der Konfigurationsvariablen

Um die Anpassungsarchitektur besser zu verstehen, haben wir die Auswahl-Konstrukte und die verwendeten Variablen näher analysiert. Hierfür haben wir das Kati-Werkzeug [5] etwas modifiziert und anschließend bei der Analyse eingesetzt. Bei Kati handelt es sich um ein Open-Source-Werkzeug von Google, das Makefiles parst und diese in Ninja-Dateien umwandelt. Mit den Ninja-Dateien kann Android dann optimiert gebaut werden, wodurch der Buildprozess deutlich beschleunigt wird - ein durchaus interessanter Ansatz. Wir haben dabei 682 Buildvariablen gefunden, die sich in 4 Kategorien einteilen lassen: Produktkonfiguration, Boardkonfiguration, Build-OS und Build-Details. Abb. 7 gibt einen Überblick, ob diese Variablen im Buildsystem oder im Quellcode zum Einsatz kommen.

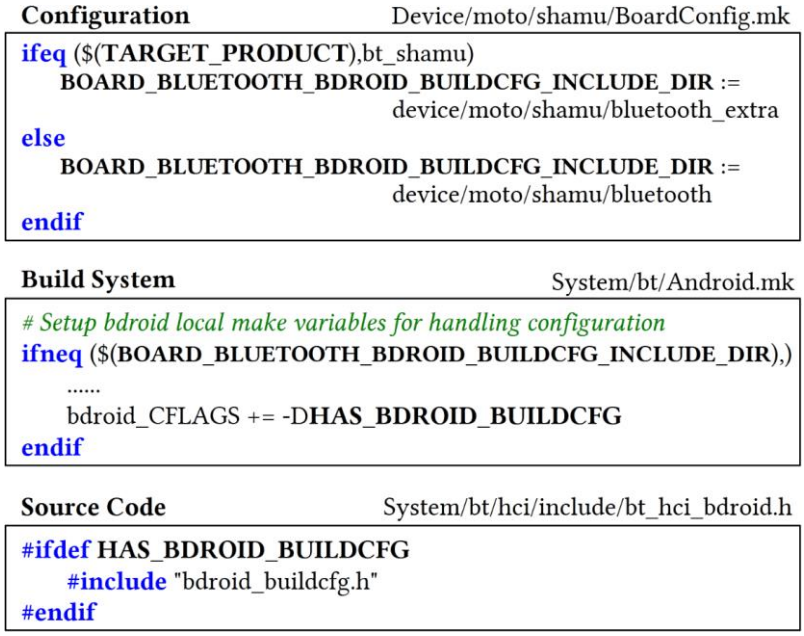

Abb. 7: Beispiel der 3-Ebenen der Anpassungsarchitektur

Anpassungen auf der Code-Ebene

Da Androids Konfigurationsvariablen in Makefiles definiert werden und nicht in Header-Dateien, stehen sie dem Präprozessor bei der Verarbeitung der Quellcode-Dateien nicht direkt zur Verfügung, sondern müssen explizit übergeben werden. Ein entsprechendes Beispiel ist in Abb. 8 dargestellt: In Abhängigkeit von übergeordneten Konfigurationsvariablen werden zunächst untergeordnete Konfigurationsvariablen gesetzt. Anschließend werden im Buildsystem in Abhängigkeit von den untergeordneten Konfigurationsvariablen Compilerflags gesetzt und darüber die Werte der Variablen an den Präprozessor / Compiler übergeben. Zunächst erscheint dies vielleicht etwas umständlich.

Bei näherer Betrachtung zeigt sich allerdings eine klare Konfigurationsarchitektur, welche die Anwendungskonfiguration klar von der technischen Konfiguration der Module separiert. Da die entsprechenden Zusammenhänge nur über die Makefiles zum Ausdruck kommen, lässt sich die Anpassungsarchitektur recht gut werkzeuggestützt analysieren, optimieren und weiterentwickeln. Hier kann man für große Projekte sicherlich etwas von Android lernen. Bei Bedarf ist das Kati-Werkzeug [5] hier sicherlich ein guter Startpunkt. Wir haben es etwas erweitert, um so an weitere Informationen zu gelangen, die bei der Analyse des Buildsystems im Werkzeug vorliegen.

Zusammenfassung

Den Android-Entwicklern gelingt es ein breites Maß an Anpassungen basierend auf einer heterogenen Codebasis, die sich nebenläufig weiterentwickelt, zu unterstützen. Damit liegt die Vermutung nahe, dass die Anpassungs- und Konfigurationsarchitektur von Android sicherlich ein sehr guter Ideengeber für Praktiker ist, die im Entwicklungsalltag mit der Variantenbeherrschung kämpfen. Da es leider nicht ganz einfach ist, sich einen entsprechenden Überblick zu verschaffen, haben wir eine entsprechende Analyse durchgeführt und in [2][3] umfassend dokumentiert.

Die Analyse führte bei uns zu folgenden Erkenntnissen: Die Anpassungsunterstützung in Android kommt deutlich einfacher daher als wir das erwartet hatten. Statt ausgewieften Konfigurationsmodellen, -techniken und –werkzeugen sowie klaren Codierstandards, kommen eher die üblichen Verdächtigen zum Einsatz. Es fällt auf, dass die 3-Ebenen-Konfigurations- und Anpassungsarchitektur sehr klar ist und sich im Wesentlichen auf Makefiles stützt. Die Konfiguration aus Anwendersicht ist strikt von der technischen Konfiguration der Module getrennt. Ein Muster, was uns auch schon an anderen Stellen begegnet ist. Die Abhängigkeiten zwischen Anwendungskonfiguration, Buildsystem und Quellcode steckt in den Makefiles und kann unter Verwendung des Kati-Werkzeuges extrahiert werden. In den nächsten Releases von Android soll Make durch Kati/Ninja ersetzt werden. Am diesem Beispiel zeigt sich auch, wie die Transition zwischen verschiedenen Buildsystemen auch in komplexen Systemen gelingen kann.

Danksagung

Die vorgestellten Analysen wurden von Nicolas Fußbender im Rahmen eines Studentenprojektes der TU Kaiserslautern am Fraunhofer IESE durchgeführt. Vasil Tenev und Dr. Bo Zhang haben die Durchführung der Arbeit maßgeblich unterstützt.

Literatur- und Quellenverzeichnis

[1] Android Device Statistics, https://www.statista.com/statistics/278305/daily-activations-of-android-devices/

[2] Nicolas Fußberger: "Analyzing the Variability Realization in Android", TU Kaiserslautern, http://wwwagse.informatik.uni-kl.de/publication/papers/Fussberger2016.pdf

[3] Nicolas Fußberger, Bo Zhang, and Martin Becker: "A Deep Dive into Android's Variability Realizations", in Proceedings of the 21st International Systems and Software Product Line Conference (SPLC '17). DOI: https://doi.org/10.1145/3106195.3106213

[4] Android Source Repository, https://github.com/android/platform_development

[5] Kati Tool, https://github.com/google/kati

Autor

Dr. Martin Becker ist Informatiker und leitet die Abteilung Embedded Systems Engineering am Fraunhofer IESE. In einer Vielzahl von Forschungs- und Industrieprojekten hat er in den letzten 17 Jahren vielfältige Erfahrungen mit Varianten-Management in verschiedenen Anwendungsgebieten gesammelt, sowie neue Methoden, Techniken und Werkzeuge auf diesem Gebiet entwickelt.

Kontakt

Internet: http://www.iese.fraunhofer.de/
Email: Martin.Becker@iese.fraunhofer.de