Variability Mechanisms in Software Ecosystems

Thorsten Berger[a], Rolf-Helge Pfeiffer[b], Reinhard Tartler[c], Steffen Dienst[d], Krzysztof Czarnecki[a], Andrzej Wąsowski[b], Steven She[a]

[a] *University of Waterloo, Canada*
[b] *ITU Copenhagen, Denmark*
[c] *University of Erlangen-Nuremberg, Germany*
[d] *University of Leipzig, Germany*

Abstract

Context: Software ecosystems are increasingly popular for their economic, strategic, and technical advantages. Application platforms such as Android or iOS allow users to highly customize a system by selecting desired functionality from a large variety of assets. This customization is achieved using variability mechanisms.
Objective: Variability mechanisms are well-researched in the context of software product lines. Although software ecosystems are often seen as conceptual successors, the technology that sustains their success and growth is much less understood. Our objective is to improve the empirical understanding of variability mechanisms used in successful software ecosystems.
Method: We analyze five ecosystems, ranging from the Linux kernel through Eclipse to Android. A qualitative analysis identifies and characterizes variability mechanisms together with their organizational context. This analysis leads to a conceptual framework that unifies ecosystem-specific aspects using a common terminology. A quantitative analysis investigates scales, growth rates, and—most importantly—dependency structures of the ecosystems.
Results: In all the studied ecosystems, we identify rich dependency languages and variability descriptions that declare many direct and indirect dependencies. Indirect dependencies to abstract capabilities, as opposed to concrete variability units, are used predominantly in fast-growing ecosystems. We also find that variability models—while providing system-wide abstractions over code—work best in centralized variability management and are, thus, absent in ecosystems with large free markets. These latter ecosystems tend to emphasize maintaining capabilities and common vocabularies, dynamic discovery, and binding with strong encapsulation of contributions, together with uniform distribution channels.
Conclusion: The use of specialized mechanisms in software ecosystems with large free markets, as opposed to software product lines, calls for recognition of a new discipline—variability encouragement.

Keywords: software ecosystems, empirical software engineering, software product lines, variability management, mining software repositories

1. Introduction

Large software ecosystems implement *variability*—the diversity of systems they offer—using radically different implementation techniques, also known as *variability mechanisms*. Consider the Linux kernel and the Android application platform for mobile devices. The former is a highly configurable system that implements variability in the source code, by conditionally compiling the desired functionality. The latter is a service-oriented architecture that encourages variability by letting users easily install new extensions (apps). Yet, both successfully facilitate, manage, and technically support a high degree of customization freedom for the intended recipients of the systems.

The Linux kernel and Android are examples of two major classes of large, highly successful software ecosystems. Linux manages variability centrally, carefully controlling the admission of new features into its official release. Android manages variability decentrally, embracing and encouraging variability within a free market of apps. Both systems rely on different mechanisms to achieve their goals.

The Linux kernel uses mechanisms known from *software product line engineering* (SPLE) [1, 2]. SPLE allows companies to efficiently create portfolios of systems in an application domain by leveraging the commonalities and carefully managing the variabilities among the systems [3]. For instance, the Linux kernel supplements conditional compilation with a *variability model*, which abstractly represents thousands of variabilities, such as drivers and processor architectures. Such models are popular means to manage variability. Despite using mechanisms of SPLE, the Linux kernel is more than a product line—it is a *software ecosystem* [4]. Around 7,800 developers from 800 companies have helped to more than double the code base from 6.6M to 15M lines of code (LOC) within seven years [5].

Android also manages huge variability, but in a more compositional and open way. Users derive a concrete system by selecting apps from online repositories using an installer tool—in effect, composing their system from third-party components (apps). In contrast to Linux, Android has no centralized and integrated variability model, but describes variability information decentralized within each app. Android is also an ecosystem, but unlike Linux's respectable, yet controlled growth, Android has virtually exploded with tremendous growth rates, offering over one million apps today.

Research has addressed software ecosystems, but focused on economic, strategic, and organizational aspects [6, 7, 8], less on technology [9]. What mechanisms are effective in practice, and in what context? What are their core characteristics? While variability mechanisms in software product lines are reasonably well researched [10, 11], their role in supporting a software ecosystem is much less understood [9]. In fact, developing models that describe ecosystems [12] and defining theories that explain concepts and causalities [13], are key research challenges in this field.

We address this gap with an exploratory study of the solutions to variability in software ecosystems. We analyze five software platforms and their surrounding ecosystems: the eCos operating system (OS), the Linux kernel project, the Debian Linux distribution, the Eclipse Integrated Development Environment (IDE), and the Android OS. Some of these are among the largest and fastest-growing ecosystems in existence today. All successfully facilitate massive variability, while approaching variability from different organizational and business perspectives and using different variability mechanisms.

Our research objectives are (**O1**) to identify and analyze variability mechanisms in ecosystem platforms, and (**O2**) to discover relationships and potential causalities among the mechanisms. We strive to understand how software ecosystems technically facilitate variability. Our study involves a qualitative analysis of the variability mechanisms used in the platforms and of the organization of their development and variability management. A quantitative analysis investigates scales, growth rates, and—most importantly—the structure of the dependencies among the variabilities in the ecosystems. We describe observed phenomena and core differences discovered through our analyses, develop hypotheses, and raise questions for future research.

We contribute: (**C1**) a conceptual framework defining key characteristics of variability mechanisms and their organizational context within and across the ecosystems; (**C2**) an instantiation of the framework with empirical data for each ecosystem; (**C3**) a set of core differences across the subjects, and hypotheses as proposed explanations; and (**C4**) static analysis tools and extracted datasets about all ecosystems for reproducibility and future research. These and more details are available in our online Appendix [14].

Our work represents the exploratory phase in the long-term process of theory building. We discover phenomena and develop hypotheses based on empirical evidence, widening our understanding of variability mechanisms from product lines to ecosystems. We hypothesize that if we understand the causalities of using a mechanism, we will be (i) able to predict how it sustains success and growth of an ecosystem, and eventually (ii) guide development and management. Our findings also generate requirements for tools, and our datasets can serve as realistic benchmarks.

We proceed with background information Section 2. We detail our study design in Section 3. We introduce our conceptual framework in Section 4, followed by instantiations of the framework with qualitative and quantitative empirical data in Sections 5–7. Thereafter, we synthesize results of our cross-case analysis and develop hypotheses in Section 8. Finally, we discuss threats to validity in Section 9, related work in Section 10, and conclude in Section 11.

2. Background

Software ecosystems is an emergent field of research that has been addressed from various perspectives. So far, researchers have not agreed on a common definition from the perspective of technology. Yet, ecosystems are often considered as technical constructs [15, 16], arguably with fluid boundaries to related paradigms, such as distributed systems or componentware [17]. We take the view of ecosystems being extensions of product lines of substantial size [4, 12, 18, 19]. We consider the following two characteristics of an ecosystem, as defined by Hanssen [20], as central to our study: (i) a network of organizations, and (ii) a common interest in central software technology.

We focus on ecosystems that rely on a common *technological platform*, in contrast to those that are purely social, strategic, or economic constructs of loosely related software assets or projects. A platform provides the basis for mass-customization. It allows consumers to derive an individual *instance*, such as a phone (Android) or an IDE (Eclipse), using an automated, tool-supported process. An instance is derived from a universe of compositional assets—the ecosystem.

2.1. Architectural Openness

Ecosystem platforms have different degrees of openness [9]. We define openness as the extent of technical support for consumers to freely use assets from an ecosystem in their instances—that is, their product instances or installations. Openness ranges between two extremes: *closed* (no support) to *open* (full support). In both cases, users can use assets from the ecosystem; however, in closed platforms, these must be integrated into the platform first. In other words, openness "is the degree to which a platform supplier allows [and supports] the platform users to interact with the platform, view, extend or change its components" [9].

Openness is a core distinguishing characteristic of our subjects. As we will see, eCos and the Linux kernel can be classified as *predominantly closed*. Their focus is on managing variability by carefully controlling contributions to the

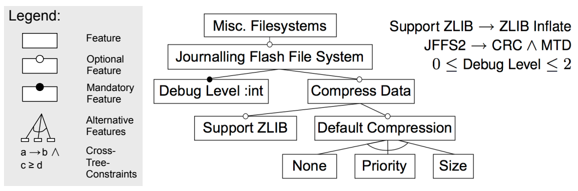

Figure 1: Feature model example (adapted from [2])

```
1   Package: gawk
2   Version: 1:3.1.7.dfsg-5
3   Maintainer: Arthur Loiret <aloiret@debian.org>
4   Depends: libc6 (>= 2.3)
5   Provides: awk
6   Section: interpreters
7   Priority: optional
8   Description: a pattern scanning and processing language
9   Architecture: i386
```

Figure 2: Excerpt of the manifest of the Debian gawk package

platform and by avoiding complexity through unnecessary variability. Still, eCos and the Linux kernel have a free market of third-party solutions, but their use requires highly technical skills from users. In contrast, Debian, Eclipse, and Android focus on encouraging variability. They are *open* by design and offer convenient facilities for users to easily extend their instances with free market assets.

Consequently, our study carefully considers the openness of the studied subjects and investigates the technical differences between the closed and open platforms.

2.2. Variability Mechanisms

A variability mechanism is an implementation technique to delay design decisions about functionalities and qualities of a software system [21, 10]. To flexibly adapt software to user requirements at later stages—such as build-, startup- or run-time—variation points are introduced into a software platform, using an architecture that supports variability. Variability mechanisms are used to implement variation points, which are locations in assets where variability occurs. Variation points are bound with concrete variants, which are either known (closed platform) or unknown (open platform) during platform development.

A range of static and dynamic variability mechanisms exists. In *static configuration*, all variation points are bound at build-time. The whole configuration of the system has to be decided before and cannot be changed at run-time. A common static mechanism is *conditional compilation* [10, 22], which is often realized with a build system that selectively compiles source files, and with a preprocessor (such as the C preprocessor) that cuts out irrelevant parts of the source file before compilation. In dynamic mechanisms, variation points are bound much later, and can usually be changed at run-time. Commonly, configuration parameters are loaded at startup and influence the run-time of the system. Thus, dynamic configuration trades static optimization of systems (e.g., memory footprint) for flexibility. Further mechanisms comprise code generators (static) and *component-oriented architectures* (usually dynamic), which load components according to a specific configuration. Even more dynamic are *service-oriented systems*, where components offer services for interaction. Loading and configuring such components and their interactions is usually fully dynamic.

To handle large numbers of variation points, product lines commonly use variability models, such as feature [23] or decision models [24]. Variability models abstractly describe and organize the variabilities at a central place, and are input to configurator tools, which help users resolving variabilities [25]. Consequently, we classify the techniques for variability models as variability mechanisms, too.

In addition to *variability models*, another approach is used in our ecosystems to represent variability information: *manifests*. To set the stage for our analysis, we briefly introduce these two ways of describing variability.

Figure 1 shows a feature model—a popular variability modeling language and notation. Our example describes the variability of the Journalling Flash File System supported in eCos and the Linux kernel. Feature models express the commonality and variability of a product line as features organized in a hierarchy. Constraints restrict their valid combinations and values. In our example, the feature Debug Level is mandatory (solid circle), whereas the ability to Compress Data is optional (hollow circle). Default Compression is a feature group that allows selecting exactly one child feature. Additional constraints are listed to the right. In contrast to a *variability model*, which describes the whole variability of a system centrally, *manifest files* can be used when variability should be described in a distributed way. A manifest file declaratively describes metadata and variability information (e.g., dependencies) of one asset, and is packaged and maintained together with it. Similar to variability models which are declared in a formal language, manifest files adhere to a schema, which can be a grammar for textual manifests, or an XML Schema for XML-based manifests. Figure 2 shows the excerpt of a (textual) Debian manifest of the GNU/awk interpreter package. It contains naming (l. 1), versioning (l. 2), dependency (l. 4–5), and categorization (l. 6–7) information. These are the typical contents of a manifest file; however, more complex descriptions of components (as in Android manifests) also occur.

Variability models and manifests are two core variability mechanisms found in our five subjects. In our study, we identify further mechanisms and their characteristics, and explore the relationship to their organizational context.

2.3. Dependencies

Interactions between assets introduce dependencies that are declared in variability models (eCos, Linux kernel) or manifests (Debian, Eclipse), or are hidden in code (Android). They are core characteristics of variability mechanisms. They complicate development and maintenance, but also challenge tools. We study dependencies to learn how our subjects cope with this complexity. We analyze how dependencies are expressed and what dependency structures exist that tools and consumers have to manage. Specifically, analyzing Android helps to understand how one of the largest and fastest-growing ecosystems tackles complexity.

3. Methodology

We perform case study research [26, 27, 28] with five cases—our subject ecosystems. The goal is to discover real-world phenomena and generate hypotheses from empirical evidence, which is the exploratory phase of theory building [26]. Generating hypotheses by analysis of case studies is a highly qualitative and interpretive process. These hypotheses need to be refuted or confirmed using other methods, such as experiments and simulations, and confronted with further data, which is future work.

3.1. Case Study Selection

Our selection of subjects strives for broad applicability of the resulting conceptual framework. We chose five successful ecosystems spanning diverse domains and approaching variability in different ways. They range from systems with central variability models and static configuration using conditional compilation, through component-oriented architectures specifying variability in separate manifest files associated with assets, to highly dynamic service-oriented systems with dynamic configuration of assets at run-time. Since our subjects and most of their assets are open source, we can study significant subsets of their ecosystem.

Each of our subjects is an ecosystem on its own, although overlaps and interactions among them exist. Each subject spans a universe of assets that is conceived or managed as an ecosystem, with individual organizational structures and communities. For instance, the Debian ecosystem consists of software packages managed by the Debian community, even though it also contains packages of Eclipse and the Linux kernel. Since the latter have their own ecosystems with dedicated communities, we clearly distinguish among them.

3.2. Qualitative Analysis

The major part of our analysis is qualitative. It focuses on identifying mechanisms and organizational structures in the studied ecosystems and relationships among them.

We followed recommended practices of case study research [28]. We first performed within-case analyses, by creating in-depth write-ups of each ecosystem. These write-ups are part of our online Appendix [14] (cf., Section 3.4). During analysis, we iteratively built a conceptual framework of the variability mechanisms and organizational structures. The framework is instrumental to compare our subjects and to unify ecosystem-specific aspects using a common terminology. The framework is summarized in Figure 3 and in the concept hierarchy shown in the left-most column in Tables 1–4. We seeded the framework with characteristics of mechanisms known from SPLE and then expanded to those specific to ecosystems. Many are inspired from literature, such as [2] (variability models, dependencies), [29] (binding time/mode, openness), and [17] (interaction, encapsulation); others were added as discovered.

Thereafter, we performed a cross-case analysis to identify the major differences across our subjects. Finally, we developed testable hypotheses to explain the observed phenomena and differences.

Our sources are referenced as we use them in the text. In the qualitative analysis, we relied on official documents, such as the Debian Policy [30] and the Eclipse Development Process description [31]. We also examined tools and languages used in the ecosystems.

3.3. Quantitative Analysis

Quantitative analyses allow us to ask questions about occurrence and frequency of identified mechanisms. It is instrumental to identify potential correlations between qualitative concepts and quantitative measures, such as growth rate of an ecosystem or dependency structures.

For the quantitative analysis, we used statically extracted data. Since analyzing whole ecosystems is infeasible given their open and uncontrolled nature, we mined substantial subsets from the most vibrant parts—the respective major distribution sources of the ecosystems. For eCos, we analyzed all i386-specific and hardware-independent packages from the repository (v. 3.0). For Linux, we studied the x86 architecture from the 2.6.32 codebase. Debian's subset are all binary i386 packages from the main component of the 6.0 distribution. For Eclipse, we analyzed the Helios 3.6 modeling distribution together with bundles from the associated repository. For Android, we gathered nearly all available free apps from Google Play over a period of 14 months in 2011 and 2012. The exact sizes of our analyzed ecosystem subsets are listed in Table 5 (first row).

We developed analysis tools for each ecosystem. For eCos and Linux, we reused and extended our previously developed infrastructure [2]. For Debian, we analyzed the package indices that are intended to be used with the native Debian package manager, and parsed manifests with a software package (python-apt) commonly used for that purpose. For Eclipse, we installed all bundles in a running system and used the platform API to query information. Analyzing Android was by far the most challenging, since dependencies are not explicitly declared. We implemented static analysis techniques for identifying Intent calls and their parameters from Android (Dalvik) bytecode.

3.4. Reproducibility

For further research, and for reproducibility of our study, we provide an online Appendix in a repository [14]. It contains the detailed within-case write-ups for each subject, our developed analysis tools, and datasets. Further statistics, diagrams, and details about all estimations (e.g., scale and growth rate of ecosystems) are available in Appendix B of [32], including details on the Android bytecode analysis.

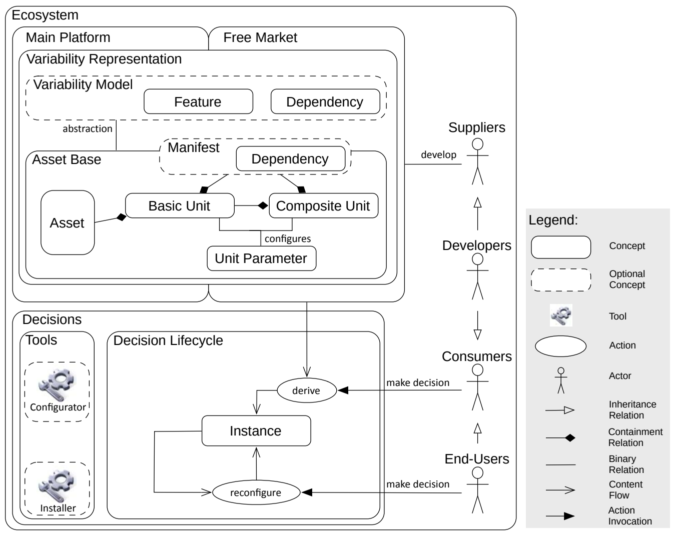

Figure 3: Illustration of the conceptual framework

4. Conceptual Framework

We describe our conceptual framework in this section, and later use it to characterize and compare the five ecosystems. In the description, general framework concepts are typeset in sans-serif, and ecosystem-specific instantiations in *cursive*. Figure 3 illustrates core parts of the framework.

A software ecosystem is a universe of shared assets centered around a common technical platform. In this universe, various roles, mainly suppliers and consumers, interact in order to develop, manage, and consume assets. More roles exist, but modeling them is out of our scope. A platform denotes the technical aspects of an ecosystem: a variability-enabled architecture, a set of shared core assets, tools, frameworks, and patterns, together with organizational and process-related concerns. Every vital ecosystem has a controlled central part, the main platform, which is managed by the platform supplier. Free market is the less-controlled, complementary part of the ecosystem that provides third-party assets extending the main platform. Alternative platforms may exist as derivatives of the main platform for specific needs. For example, Ubuntu is a Debian derivative derived from the Debian main platform for desktop and laptop users. Since derivatives do not belong to the free market, we decided to ignore them in this study.

Assets are any artifacts, such as source code, binaries, media files, or documentation. Each of the studied platforms packages assets into basic units, such as *Debian packages* or *Eclipse bundles*. Composite units, such as Debian *meta packages*, aggregate sets of basic units.

Variability in the platforms has two forms: basic units can be optional, or vary inside, or both. Unit parameters, such as *properties* in Eclipse, describe variability within basic units.

An instance (e.g. a customized Linux kernel or a particular installation of an Android system) is a concrete system derived from the main platform and the free market by making decisions—more precisely, by selecting and configuring assets, thus, resolving variability. Usually, an instance can be re-configured later.

Variability information (dependencies and unit parameters) is specified either within a variability model or in distributed manifests. Variability models are system-wide and integrated abstractions over the concrete assets and declare features and dependencies using a dedicated language [2]. Features are abstract entities that are mapped to units and unit parameters. Instead of making decisions directly on the assets, derivation is based on deciding features. Manifests directly reflect variability information of the assets, without the ability to introduce abstractions. Such abstraction is only partially possible by using empty assets whose manifests aggregate dependencies, like Debian *virtual packages*.

Each ecosystem supports derivation and re-configuration by automated tools: configurators for the variability model-based platforms (eCos, Linux) and installers for manifest-based platforms (Debian, Eclipse, Android). Such automated tools assist consumers with intelligent choice propagation, conflict resolution, and optimization based on the dependencies. The latter are declared either among features within the variability model, or among basic or composite units within manifests.

5. Context of Mechanisms

Our ecosystems span fairly different domains, organizational structures, and achieved different scales over time. We discuss these aspects to put the variability mechanisms into context. Tables 1 and 2 summarize our observations.

5.1. Platform Domain and Target Audience

eCos is a free real-time OS for deeply embedded applications—a domain that requires high portability, low memory usage, and small binary images. With a market share of 5–6%, it powers, among others, multimedia, networking and automotive devices [33]. Consumers of eCos are highly specialized developers of embedded systems. eCos maintains advanced tools, such as a configurator with a reasoning engine.

The **Linux** kernel is a free general OS kernel targeting a much broader range of hardware than eCos. Its consumers include Linux distributors, who customize and release specialized kernels, and technically skilled end users, who sometimes also configure, compile, and install a custom kernel. The Linux kernel also provides a configurator, but much less advanced [2] than the one of eCos.

Debian is a complete OS with a large selection of applications. It is available for many hardware architectures, ranging from embedded systems to high performance computers. Its consumers are both non-technical end users and system administrators with deep expertise. Debian provides suitable installers and configurators for beginners and experts. We chose Debian as it is one of the most popular, established, and accessible Linux distributions [34].

Table 1: Ecosystem domains and organization

		eCos	Linux kernel	Debian	Eclipse	Android
Domain	Software domain	embedded OS	general-purpose OS kernel	OS & application software	software development tools	OS & applications for mobile devices
	Consumer skills	highly technical	highly technical	technical and non-technical	technical	non-technical
Organization	Main Platform	free eCos edition	mainline kernel	Debian Archive ('main' section)	yearly official platform release	Android OS and Google Apps
	Development	centralized	distributed	distributed	distributed	distributed
	Variability mgmt.	centralized	centralized	distributed	distributed	centralized
	Contribution filtering	strong filtering	strong filtering	little filtering	strong filtering	strong filtering
	Free market	packages	kernel modules (drivers), patches	mostly commercial packages	bundles on update sites/market places	apps on market places
	Distribution channel	none	none	marginal third-party repos.	Eclipse Marketplace	Google Play store
	Role of contributions	marginal	complementary	complementary	complementary	essential

The **Eclipse IDE** is a foundation for highly customizable development tools.[1] Eclipse was explicitly conceived as an ecosystem [35] and advertised as such by the Eclipse Foundation [36]. Although users of the Eclipse IDE are technically skilled developers, extending the system is supported by a convenient installer.

Android is a free OS for mobile devices, including smartphones, tablets, and netbooks, which can be extended with third party apps. The target consumers of Android are non-technical end users, deriving their system by installing apps with a user-friendly installer.

5.2. Organization

We identified the following organizational structures of development and variability management (see Table 1).

eCos' main platform is its free edition, maintained and developed by the main supplier eCosCentric and external contributors [37]. Both development and variability management are *centralized* in the main platform. We have not found reliable information about the process used for contributions (eCos packages and patches). However, the main platform is controlled by a group of currently ten maintainers, which indicates that contributions have to pass their reviews. Only a marginal free market emerged on the fringe of the main platform, although eCos' packaging mechanism and its modular variability language were designed to encourage contributions. No uniform distribution channel exists for the free market.

Linux' main platform is the mainline kernel. The variability management is *centralized*, with only a few maintainers controlling the variability model [38]. In contrast, the development is highly *distributed*, comprising thousands of developers and maintainers. However, contributions (patches, usually with new features) have to pass thorough reviews through the maintainer hierarchy. Although no uniform distribution channel (beyond mailing lists, such as the official Linux Kernel Mailing List) outside the main platform exists, an unorganized free market with third-party modules (mostly drivers) emerged.

[1]Eclipse also provides the Rich Client Platform for building arbitrary GUI software, but we focus on the IDE ecosystem.

Debian's main platform is the central repository containing the official distribution. Both development and variability management are *distributed*, comprising over thousand package maintainers, who maintain packages that are sourced from free and open source software [39]. The main platform tries to be as inclusive as possible, with little restrictions to contributions (Debian packages), while reviews still assure quality [30]. A free market with mostly commercial and non-free packages in scattered third-party repositories complements the main platform.

Eclipse's main platform is represented by the yearly releases of the IDE. It consists of independently managed projects following the Eclipse Development Process [31] and is controlled by its supplier, the Eclipse Foundation. Contributions (new projects) undergo thorough reviews. Both the development and variability management is *distributed* in the main platform. Eclipse has a complementary free market, mainly represented by the Eclipse Marketplace [40] and further repositories, such as Yoxos [41] and smaller update sites for Eclipse's installer.

Android's main platform comprises the OS and pre-installed apps. While the development is *distributed*, the variability management of the main platform is *centralized* and fully controlled by Android's supplier, the Google-led Open Handset Alliance. Individual sub-projects exist, each having a project lead (typically a Google employee [42]). Contributions (patches, possibly also apps) to the main platform are possible, but with thorough reviews. A free market is an essential goal of Android. The main distribution channel (Google Play store) is widely open for third-party contributions of arbitrary applications.

5.3. Scale and Growth

We conservatively estimated main platform and free market sizes (see Table 2). We chose LOC as our primary measure to account for the different granularities of assets in the ecosystems. LOC is also known to be highly correlated with complexity, development, and maintenance effort [43].

eCos has the smallest main platform, comprising only 502 packages and a marginal free market. **Linux** is much larger, given its support of a much wider variety of hardware. We could not estimate the possibly large, but unor-

Table 2: Estimated Scales and Growth Rates

	eCos	Linux	Debian	Eclipse	Android
Main platform scale[7]					
Basic Units	3948[1]	25,861[1]	28,232[2]	5,787[3]	83[4]
Features	2,859	10,415	N/A	N/A	N/A
LOC	0.9M	7.9M	762M	21.2M	1M
Free market scale[7]					
Basic Units	>1,530[1]	—	>15,179[2]	>1,897[3]	>651K[4]
Features	>315	—	N/A	N/A	N/A
LOC	>279K	—	>410M	>6.9M	>1G
Growth rates[7]					
Inception year	$1999^{(v1.1)}$	$1991^{(v0.01)}$	$1996^{(v1.1)}$	$2001^{(v1.0)}$	2008
Inception LOC	76k	10k	13M	141k	1.128M[5]
Current LOC[6]	1.2M	7.9M	1.2G	28.1M	1G
Growth per year	32%	39%	35%	80%	353%

[1] Files [2] Packages [3] Bundles [4] Apps [5] Android OS and apps
[6] Of considered version [6] As of 03/2012 [N/A] Not applicable
— Data not available [LOC] Lines of Code

ganized free market. **Debian** has the most inclusive and largest main platform in our study. It is relatively easy to contribute new packages. As a result, the free market [44] is comparatively small, half the size of the main platform. **Eclipse's** main platform and free market are both of medium size, compared to the others. The main platform (Helios 3.6) is three times larger than the two main free market repositories [40, 41]. However, the whole free market may be significantly larger, as the ecosystem is heavily scattered with smaller update sites. **Android** has a free market that is over 1,000 times larger than the main platform [45]. The main platform, which is relatively closed and strongly filters outside contributions, is very small with 83 apps (Android 2.3.4).

Finally, we estimated yearly growth rates of our subjects by fitting an exponential growth function to the size difference between initial release and current state. As shown in Table 2, Eclipse and Android, which strategically foster a free market, grew considerably faster than the others, which focus on the main platform.

6. Variability Mechanisms

In our study, we identified and characterized variability mechanisms both from a technical (how instances vary) and a consumer perspective (how and when consumers make decisions). Table 3 summarizes our observations.

6.1. Variability Representation

Asset Base. In **eCos**, basic units are source files with internal variability controlled by preprocessor symbols (unit parameters) and realized via #ifdef statements. Composite units are packages, which are aggregations of source files, test cases, or other resources, together with a variability model of the package. eCos' configurator aggregates partial models into a single whole, depending on the set of loaded packages. A feature-to-code mapping (declared in the model) connects features with implementation assets; it is used to derive a concrete instance. **Linux** has two types of basic units: (i) source files with preprocessor symbols (unit parameters) as in eCos, and (ii) loadable kernel modules that extend Linux at run-time. No concept for composite units exists. The feature-to-code mapping resides in the build system [46, 47]. **Debian's** basic units are packages—file archives with helper scripts and a manifest. Composite units are realized by meta packages, whose purpose is to aggregate other packages via dependencies. The tool debconf realizes unit parameters and is used by scripts to configure the packaged software. It prompts users to make configuration choices during package installation. **Eclipse's** basic units are OSGI bundles—dynamically loadable modules tying together artifacts such as Java classes, images, configuration files, and metadata. Bundles run in a virtual machine. Unit parameters are provided by several mechanisms, including the preference store and configuration admin service. Composite units called "features" aggregate multiple bundles with branding and update information. **Android** is composed of apps—individual application programs representing basic units. Most apps run in a virtual machine (Dalvik). All apps are treated equally by the virtual machine, which allows alternative implementations even for pre-installed (main platform) apps. Android has no concept of composite units, but has a dedicated mechanism for unit parameters (preferences). Android offers API support to create a unified user interface for app preferences, and to store and load them.

Variability Model. eCos and the Linux kernel come with feature-model-like variability models declared in their respective languages CDL and Kconfig. Interestingly, while Kconfig has no modularization support beyond a simple file include statement (source), CDL was designed to encourage contributions and allows a modularized specification of models, distributed over individual eCos packages.

Manifest. Debian, Eclipse, and Android have no variability model. Variability information is declared in a text- or XML-based manifest file inside a packaged basic unit, and maintained together with it.

For further details on the modeling languages and manifests, we refer to [2, 48, 49] (eCos and Linux kernel), [50] (Debian), [51] (Eclipse), and [52] (Android), in addition to our within-case write-ups (cf., Section 3.4).

Grouping and Categorization. To be usable by consumers, units and features need to be organized in some form. eCos and Linux organize features hierarchically in variability models [2], whereas units are organized in diverse, often informal, ways in the other systems. Variability models use the hierarchy to group and categorize features. Abstract features, which are not mapped to code, improve the structure, but can also be used to optimize dependencies [53]. In the other subjects, public repositories, such as the Eclipse Marketplace [40] and Google Play [54], have their own categorization systems. Debian also offers community-driven categorizations using Debtags [55].

Table 3: Variability mechanisms

		eCos	**Linux kernel**	**Debian**	**Eclipse**	**Android**
Variability Representation	Asset Base					
	Basic units	files	files, kernel modules	packages	bundles	apps
	Composite units	packages	N/A	meta packages	features	N/A
	Unit parameters	preproc. symbols	preproc. symbols	debconf options	properties/ preferences	preferences
	Variability model	feature-model-like	feature-model-like	N/A	N/A	N/A
	Features	packages, components, options, interfaces	configs, choices, menuconfigs, menus	N/A	N/A	N/A
	Language	CDL	Kconfig	N/A	N/A	N/A
	Manifest (Schema)	N/A	N/A	y (textual DSL)	y (OSGI manifest)	y (XML-based DSL)
	Grouping and categorization	variability model	variability model	tasks, sections, debtags	market place categories	app store categories
Decisions	Decision lifecycle	derivation	derivation, re-config.	re-configuration	re-configuration	re-configuration
	Decision binding	static	static & dynamic	dynamic	dynamic	dynamic
	Derivation/re-config. tools	configurator (ConfigTool), build system	configurator (Kconfig), build system (Kbuild)	installers (apt, dpkg)	installer, market place client (P2)	installer app (e.g., Market)
Encapsulation	Interface mechanisms	C header files	C header files	package-specific	Java interfaces and OSGI manifest	explicit public components, predef. data formats
	Interface specification	documented interfaces for components, e.g., drivers	documented interfaces for components, e.g., drivers	package-specific, documented policies for some domains	explicit public interfaces defined by OSGI manifest	explicit public components, predef. data formats
Interactions	Managed by run-time system	N/A	N/A	N/A	Equinox OSGI	Dalvik VM
	Interaction mechanisms	static linking	static & dynamic linking	dpkg-triggers, documented policies	class reference, services, extension points	intent mechanism
	Interaction binding	early static	early static & dynamic	not specified	late static & dynamic	late dynamic
	Platform openness	predominantly closed	predominantly closed	open	open	open

N/A Not applicable

6.2. Decisions

The most distinguishing characteristics of decisions we identified are their lifecycle, binding, and tool support.

Decision Lifecycle. The *decision lifecycle* characterizes when and how end users decide the presence or absence of units—whether they derive an instance from scratch, or only re-configure one. In eCos and Linux, users derive an instance. In the others, users normally re-configure an initial instance provided by the supplier. Eclipse comes in one of eleven pre-instantiated editions. An Android instance is delivered with the mobile device. A Debian user usually installs a minimal system before it can be re-configured by installing and removing packages.

Decision Binding. Decisions can have different *binding mode* and *binding time*. Binding mode characterizes whether a decision can be changed. For eCos and Linux, it is static, since these systems require to re-derive the instance for changes. However, Linux also allows late dynamic decision binding by means of loadable kernel modules. Debian, Eclipse, and Android are dynamic as they allow basic units or composite units to be installed and removed at run-time.

Tools. Our closed platforms, which are mostly statically configured, provide configurators to support the derivation process. Our open platforms include an installer that allows end users to extend their instance. Both configurators and installers, except the one of Android, offer choice propagation support and reasoners to resolve dependencies between features or basic units. Android's installer does not enforce dependencies statically. Instead, apps have to handle unsatisfied dependencies at run-time.

6.3. Encapsulation

Our closed platforms offer no encapsulation concepts beyond C header files; only implementation guidelines for interfaces of loadable kernel modules exist in Linux. In Debian, interfaces are solely package-specific; however, Debian has policies for some domains, such as Java libraries or Emacs extensions. Eclipse encapsulates all classes and resources in the bundle; public functionality (Java packages, OSGi service interfaces, extension points) must be declared in the manifest. Android apps can provide public components that are described and advertised to other apps with intent filters (explained shortly in Section 7.1).

6.4. Interactions

Interactions among basic units require identifying and binding the concrete target. We identified the following interaction binding mechanisms.

eCos and Linux use static interaction binding: all selected basic units are linked into a single binary image. Linux also supports late dynamic interaction binding

Table 4: Dependencies

Dependencies	eCos	Linux kernel	Debian	Eclipse	Android
Direct dependency					
Target	features	features	basic units	basic units	basic units
Types (hard/*soft*)	hierarchy, requires, active_if, *default*, calculated	selects, prompt condition, *default*	depends, pre-depends, *recommends*, breaks, conflicts, *suggests*, *enhances*	Require-Bundle	explicit intent
Capability-based dependency					
Target	CDL interfaces	N/A	virtual packages	Java packages, services	intent filters
Types	same as direct dep.	N/A	same as direct dep.	Import-Package, dynamic service lookup	implicit intent
Common vocabulary	N/A	N/A	N/A	via API	via API
Provide capabilities	implements	N/A	provides	Export-Package	via intent filter
Expressiveness	any Boolean; arithmetic & string operations	any Boolean; number/string equality	any Boolean; version comparison	conjunction & implication; version comparison	N/A

N/A Not applicable

through kernel modules. In Debian, interaction binding is mostly package-specific, however, several policy documents prescribe guidelines for interaction in some domains. In contrast, the open platforms Eclipse and Android both provide a virtual machine that has full control over interactions. Eclipse offers three facilities: direct class referencing, extension points, and services. Except for services (using the Service Activation Toolkit or declarative services), interaction targets are bound late but statically—due to Java classloader restrictions. Android provides a purely dynamic facility for interaction with its intent mechanism. The interaction target—specified by parameters of an intent—is continuously reevaluated at run-time and could easily change when apps are exchanged or reinstalled.

6.5. Openness

Our analysis of mechanisms so far allows us to detail our characterization of the subjects' openness. We can see that openness is primarily reflected in the decision lifecycles (derivation and reconfiguration) and tools. We classify the platforms of the Linux kernel and of eCos only as *predominantly* closed. In the Linux kernel, additions need to be applied to the source tree, for example, as git branches or patch sets. This "out-of-tree" development is actively discouraged [56], and deriving such an instance is not supported by the configurator. Exceptions are loadable kernel modules from commercial vendors, and kernel derivatives offered by official Linux distributions. In eCos, although openness was a primary goal of its packaging mechanism, adding third-party functionality to an instance still requires programming effort. The variability mechanisms of Debian, Eclipse, and Android are intentionally *open*, with installer tools supporting free markets of assets.

7. Dependencies

In our study, we identified the following mechanisms to express dependencies in the platforms, and the resulting dependency structures in the ecosystems.

7.1. Specification, Semantics & Expressiveness

Our ecosystems approach specifying dependencies in diverse ways. Table 4 summarizes the core characteristics. We now discuss the declaration of dependencies, their target (units, features, or capabilities), their semantics (modality), and the corresponding constraint languages.

eCos and Linux declare dependencies among features in their variability models. Due to their high level of abstraction, variability models allow flexible specification of intricate dependency structures. This flexibility comes at the cost of maintaining additional artifacts—variability model [57] and feature-to-code mapping [46], which need to be coordinated. Debian's and Eclipse's specification of dependencies among basic units in manifests is more direct, but less flexible. Android approaches the problem entirely dynamically. No static specifications of dependencies among apps are used. Apps can only declare to be open for interaction by setting a flag, or defining an intent filter, stating that the app can handle specific service requests.

We identified a special class of dependencies in each ecosystem: dependencies on *capabilities*, as opposed to direct dependencies. Capabilities are abstractions over functionality provided by one or more units or features. For example, the capability to open URLs of a specific format (starting with "http://") is provided by multiple web browsers. All platforms except the Linux kernel provide explicit capability constructs. As can be seen in Table 4, capabilities are features in eCos and the Linux kernel, labels in Debian (such as in l. 5 of Figure 2), and mainly labels in Eclipse (with the exception of rarely used services). Android provides the richest specification via intent filter.

Intent filter can be characterized as follows. They form a simple Domain-Specific Language (DSL) or an ontology, which can be used by contributors to increase reuse. The main elements of an intent filter are action keys, category keys, and data specifications (in a URI format [58]). To interact with functionality described by an intent filter, apps throw an implicit intent by instantiating an Intent

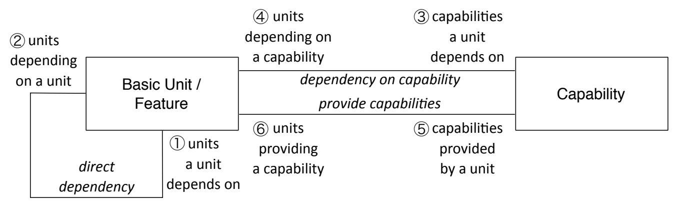
② units depending on a unit
④ units depending on a capability
③ capabilities a unit depends on
Basic Unit / Feature
dependency on capability
provide capabilities
Capability
① units a unit depends on
⑥ units providing a capability
⑤ capabilities provided by a unit
direct dependency

Figure 4: Dependency metamodel

object parameterized with an action key, category key, and a data field (URI). The intent is matched against intent filters of installed apps, which generally succeeds when an intent's parameters are a subset of the information specific in an intent filter. Thus, intents can be seen as a minimal, and intent filters as a maximal specification of functionality. A core aspect of the intent mechanism is its vocabulary. Action and category keys are provided by the Android API, but every app can introduce their own. Such third-party vocabulary needs to be documented and published for other apps to use functionality described by the vocabulary. Interestingly, the Android community has launched repositories with additional vocabularies, such as OpenIntents [59].

To abstract the dependency types we found in the languages, we created a metamodel that details the roles that units, features, and capabilities can play in dependencies. It is shown in Figure 4. A unit or a feature can directly depend on other units or features (①), or it can depend on capabilities (③). The metamodel also captures the reverse directions of these two dependency types (② and ④). The relationship that capabilities are provided by basic units or features (⑤), and its reverse (⑥), are likewise captured.

We also classified the dependencies by their semantics (modality). Hard dependencies must always be satisfied. Soft dependencies represent suggestions or defaults. We even observed conditionally hard or soft dependencies (defaults in Kconfig) that assume a different modality depending on a side condition. Table 4 (rows "Types") shows the keywords in the variability languages/schemas declaring a certain type of dependency. Notably, Debian provides the richest set of modalities, mainly to drive its sophisticated package update, replacement, and removal processes.

The constraint languages for declaring dependencies differ in expressiveness. eCos' CDL supports most operators of a modern programming language [2]. Kconfig supports any Boolean dependencies and equality on strings and numbers. Notably, it uses three-state logic for dealing with loadable kernel modules [2]. Debian supports any Boolean dependencies among packages and comparisons on version ranges. Exclusions are specified via the modalities conflicts and breaks, and defaults via recommends. Eclipse supports implications, conjunctions, and version comparisons, but lacks negations and disjunctions. It is not easily possible to exclude bundles or declare alternatives.

Table 5: Dependency Statistics

	eCos	**Linux**	**Debian**	**Eclipse**	**Android**
Ecosystem subset					
Basic units	1023[1]	10,326[1] 2,814[2]	28,232[3]	2,105[4]	281,079[5]
Features	1,244	6,308	N/A	N/A	N/A
LOC	302K	4,3M	782M	7,8M	433M
LOC per basic unit[†]	295	416	27,699	3,705	1,539
Basic units/features					
With dependencies	99%	100%	96%	89%	69%
direct①	99%	100%	95%	81%	14%
to capability③	8%	—	24%	27%	68%
With depending units②	42%	31%	62%	57%	—
Providing capability⑤	10%	—	13%	80%	100%
Dependencies ①③					
# per basic unit/feature[‡]	1	2	4	6	1
Capabilities					
With depending units④	44%	—	54%	11%	—

[1] Files [2] Loadable modules [3] Packages [4] Bundles [5] Apps
[†] Average [‡] Median ◯ Numbers refer to metamodel (Figure 4)
N/A Not applicable — Data not available (limitation of analysis)

7.2. *Dependency Structures*

We now quantitatively analyze the occurrence of the identified types of dependencies in substantial subsets of the ecosystems. To study dependency structures, we computed cardinalities for all association ends in our dependency metamodel shown in Figure 4. Table 5 shows detailed numbers. We cannot give reliable numbers on capability-based dependencies for Kconfig, since the language lacks a concept for capabilities. For Android, due to limitations of our analysis (cf., Section 9.2), we cannot reliably calculate the reverse directions (② and ④) of dependencies. In the following, we discuss the connectivity (the extent to which units or features are connected) and the density (to how many others units or features are connected) of an abstracted dependency graph.

7.2.1. *Connectivity*

The connectivity of the dependency graph indicates the proportion of units and features for which dependency information has to be maintained. The number of units or features having direct (① in Figure 4) and capability-based (③) dependencies is surprisingly high, regardless of platform openness and existence of variability models. The highest is observed in Linux, where almost all features reference others, and in eCos, where it reaches 99%. These numbers are high, partly because every non-root feature implies its parent in the model hierarchy. Still, many features (30% in eCos, 85% in Linux) declare cross-hierarchy dependencies. These are known to critically influence hardness of reasoning both for configuration tools [60] and for users, by introducing intricate implications of choices. Finally, in the open systems, most basic units also participate in many dependencies: Debian has the highest amount with 96%, followed by Eclipse with 89%, and Android with 69%.

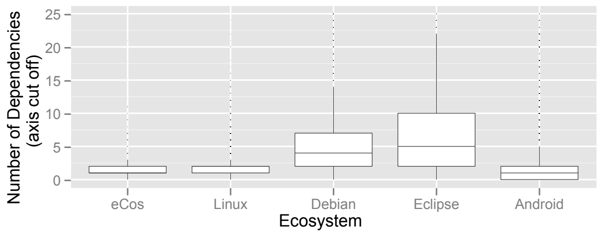

Figure 5: Dependencies per feature or basic unit

In summary, across all systems, tools supporting variability, including configuration, derivation, and analysis, must handle large numbers of dependencies.

7.2.2. Density

The density of the dependency graph indicates how much dependency information needs to be maintained per unit or feature. To assess it, we considered the number of dependencies of unit/features and of capabilities—that is, cardinalities of all association ends in the metamodel. We now discuss these numbers in general, and at the end provide a separate discussion on the extent of capability-based dependencies.

Overall Density. We first considered the number of dependencies per unit or feature in the forward direction (① and ③). Figure 5 shows the distribution of these numbers across our subjects. Surprisingly, except Android, the open platforms have in average more dependencies per unit than the others per feature. However, we also find many outliers, such as an Android app with 96 dependencies, a Debian package with 323 dependencies, and an Eclipse bundle with 419 dependencies. Some Debian outliers have many soft dependencies (modalities like *suggests* and *recommends*), indicating their importance for package installation and update processes. Still, this only happens for outliers; the majority of dependencies is hard in Debian. Finally, while many Eclipse outliers, such as the one with 419 dependencies, are caused by many Java package imports (capability-based dependencies), most dependencies are direct ones on bundles.

We also investigated the reverse direction of dependencies (② and ④ in Figure 4). If units have many, they are particularly hard to evolve, since dependencies on them are not specified directly together with the unit, but are scattered over the whole ecosystem. A developer has to know all depending units and carefully evolve it. Evolution of such units can break dependencies easily. We obtained numbers for all systems except Android, given our analysis limitations. We find that the open ecosystems have higher proportions of units being referenced (Debian: 62%, Eclipse: 57%) than the others for features (eCos: 42%, Linux: 31%). We further notice that in Debian, 44% of packages depend on one single package, `libc6`, and in Eclipse, 58% of bundles depend on `org.eclipse.core.runtime`. In the other subjects, we could not observe such an outstanding central unit or feature (maximums are 4% and 8% of features depending on a specific feature in eCos and Linux kernel).

Capability-based Dependencies. All ecosystems use capability-based dependencies. Interestingly, as can be see in Table 5, the percentage of units or features with direct dependencies drops significantly from eCos with 99% to Android with only 14%. The opposite is observed for capability-based dependencies, which rise from 8% in eCos to 68% in Android. Recall that we have no numbers for the Linux kernel, since the Kconfig language has no explicit capability construct. However, we found that some features in the Linux model play this role, using the convention to prefix them with HAVE_ (e.g., `HAVE_IDE`).

8. Phenomena and Hypotheses

Having analyzed our subjects in-depth, we now summarize their core differences related to variability mechanisms. For each difference, we propose explanations and develop hypotheses. We also identify open research issues. According to our conceptual framework, we begin with the contexts in which the mechanisms are used. We then compare characteristics of the mechanisms, including dependency declaration facilities and actual dependency structures. As architectural openness is a core distinguishing characteristic of our subjects, our comparison focuses on this aspect.

8.1. Context of Variability Mechanisms

The domains range from systems software (eCos, Linux kernel), which requires highly technical skills from users, to consumer-oriented mobile apps (Android). We learn that the organizational structures of variability management and development are independent. While all ecosystems foster distributed development, the variability management activities are performed centrally in the closed, and in a distributed way in the open platforms. We identify different processes for contributions to the ecosystems. The closed platforms strongly filter contributions using heavyweight processes including manual reviews; the open platforms use lightweight processes (little filtering of outside contributions) in uniform distribution channels. We observe highly diverse growth rates, with a clear gradation from eCos to Android. This illustrates the significance of variability encouragement in the open platforms.

8.2. Variability Mechanisms

8.2.1. Variability Representation

Differences. Variability is represented differently across the closed and open platforms. The core differences lie in the abstraction of variability and the distribution of variability information. Our closed platforms rely on centralized variability models expressed in rich languages, while our open platforms use distributed manifest files. We found that a clear difference between manifests and variability models is that manifests are always fully distributed, created as individual units with bilateral relations to other manifests, and used and evolved as individual units. In contrast, variability models, even if split over multiple files, are created around a central hierarchy, and used and evolved as an integrated whole. Furthermore, the granularity of variability differs. The ecosystem tends to comprise very fine-grained basic units in the closed, and rather coarse-grained ones in the open platforms.

Proposed explanation. Variability models are effective in centralized variability management. Features abstract over codebases and variation points. Their rich languages and the arbitrary asset mapping enables fine-grained variability and almost arbitrary cross-cutting contributions, which occur in the closed platforms. Consider `CONFIG_SMP`, a Linux feature that enables the kernel to operate on multiprocessor machines. The implementation of `CONFIG_SMP` cross-cuts the kernel code, affecting central design aspects such as handling locks, or interrupt and trap handlers.

On the other hand, variability models are ineffective in a distributed setting. Fine-grained variability and cross-cutting features requires thorough and centralized governance of the model to prevent corruption. Changes to cross-cutting features have far-reaching implications due to complex dependencies. Thus, they should be done carefully. The advantage of a variability model is that it creates a shared vocabulary to express cross-cutting properties. However, standardization of names is harder to achieve in a distributed setting. Similarly, the feature hierarchy requires more coordination than changing flat distributed variability descriptions, like in manifests. We also find very expressive constraint declaration facilities in variability models, as opposed to manifest files. These findings together indicate:

HYPOTHESIS 1. *A centralized variability model is too fragile for distributed variability management.*

Many developers can contribute code and changes to the variability model. However, a small team has to watch the impact of changes to prevent corruption.

To facilitate distributed variability management, our findings indicate that platforms need to rely on manifest files, which usually have less-expressive dependency facilities describing coarse-grained, non-cross-cutting variability:

HYPOTHESIS 2. *Distributed variability management relies on distributed variability information via manifests.*

The opposite direction of this hypothesis does not have to hold. eCos has a centrally managed, but distributed variability model (via eCos packages). Since eCos failed to create a vibrant free market, there is so far no evidence that distributed variability management could work when variability is described in a distributed variability model, richer than simple manifest files.

8.2.2. Decisions

Differences. The decision binding focuses on early static binding in the closed platforms, and tends towards late dynamic binding in the open ones. Eclipse and Android are most dynamic, realizing the binding in virtual machines controlling the run-time of basic units. The decision lifecycle also differs. The closed platforms focus on derivation of complete instances using configurator tools, while reconfiguration is the typical approach in the open platforms using installer tools. Tools differ in their support for dependency resolution. All tools support choice propagation and conflict resolution, except Android's installer.

Proposed explanation. We discuss the decision binding in the next two subsections. The other differences can be explained by the target audience of the platforms. Full derivation of instances is problematic for the less technical users of open platforms, which therefore focus on reconfiguration. Missing dependency resolution in Android can be explained by missing constraint declaration facilities in the manifests—currently, the Google Play installer would have to implement (unreliable) dataflow analyses.

8.2.3. Encapsulation

Differences. We have not observed any encapsulation concepts—such as interfaces (beyond header files)—in the closed platforms. In contrast, Android and Eclipse provide strong interface definition facilities. Debian also lacks strong encapsulation concepts, but provides policies (conventions) for some package domains.

Proposed explanation. Variability mechanisms with late dynamic decision binding require encapsulation concepts, which provide run-time guarantees about the behavior of basic units. Such guarantees cannot be assured earlier, as in the closed platforms. In turn, encapsulation concepts are not applicable in our closed platforms, due to the fine-grained variability and the cross-cutting nature of features. Instead, we can see that the run-time guarantees of encapsulation concepts are compensated by the rather heavyweight contribution processes in the closed platforms.

8.2.4. Interactions

Differences. Similar to the decision binding, the interaction binding differs across the closed and open platforms. The former have less dynamic interactions between basic units than the open ones, with a few exceptions (loadable kernel modules in Linux). Further, interactions in Eclipse and Android are controlled by a virtual machine.

Proposed explanation. The need for dynamic adaptations can be explained by the ecosystem domains. However, we can also see that the increased degree of controlled interactions at run-time (Eclipse, Android) is counter-balanced by heavyweigt contribution processes to assure quality in the other platforms (eCos, Linux kernel, Debian). Together with our insights from the decision binding and the encapsulation concepts, we hypothesize:

HYPOTHESIS 3. *Missing encapsulation and interface concepts need to be compensated with heavyweight contribution processes to assure run-time guarantees.*

On the other hand, heavyweight contribution processes would negatively impact the goal of encouraging variability in the open platforms.

8.3. Dependencies

Differences. The variability mechanisms in our subjects encompass diverse facilities to express dependencies. We found very expressive constraint languages in the closed, and relatively simple ones in the open platforms. Debian is again in the middle of this spectrum.

One of our most interesting findings are capability-based dependencies, which target abstractions of functionalities—capabilities—instead of basic units or features. Contrary to the expressiveness of dependencies, the facilities to describe capabilities and capability-based dependencies increase in their expressiveness towards Android. An important aspect of capabilities is their vocabulary. The main vocabulary is always controlled within the main platform; however, our open platforms also support third-party vocabularies in the free market.

Proposed explanation. We see a relation of the expressive constraint languages to the granularity of variability and the early static decision binding, as identified before. These are typical requirements of a technical domain:

Hypothesis 4. *In systems software, dependencies need to be more expressive than in end-user applications due to the need for low-level, fine-grained, and static configuration.*

We are, however, unaware of any study that explains this complexity by performing a systematic requirements analysis and linking the requirements to dependencies.

Fast-growing and large-scale ecosystems require constructs that can precisely describe the semantics of a capability. Labels are too ambiguous for this purpose—although they can be constructed to be unique, as seen by Debian capabilities. These range from simple (e.g., `x-window-manager`) to intricate (e.g., `libghc6-agda-dev-2.2.6-8c324`) labels. DSLs like Android's intent filter are a more accurate and viable description of abstracted functionality. However, what language constructs in capability DSLs are suited for software ecosystems, beyond Android's intent mechanism, remains a research question.

8.4. Dependency Structures

Differences. The median of declared dependencies per basic unit is higher in the open systems than the dependencies per feature in the closed systems. This phenomenon is seen across the subject spectrum except Android, which does not declare dependencies. Capability-based dependencies are essential and used in all ecosystems to varying extents, even if the platform has no explicit concept for capabilities. Our open and dynamic subjects have a significantly higher proportion of capability-based dependencies.

Proposed explanation. Variability models impact dependency structures, since dependencies are specified over abstract entities (features) mapped to the physical assets (basic units). Variability models let developers optimize and collapse implementation-level dependencies, while the coordination cost for these activities in a distributed setting may be too high. This characteristic of variability models leads to a lower density of dependencies:

Hypothesis 5. *Centrally managed variability using variability models facilitates sparse dependency structures.*

Still, there can be other reasons for the lower number of dependencies in the systems with variability models. Thus, this possibly controversial hypothesis requires confirmation.

The higher ratio of capability-based dependencies in the open platforms can be explained by their ability to reduce coupling (targets can be exchanged easily). They also improve flexibility and communication among developers, as they indicate that specific functionality is available. Capabilities are also abstractions over functionality that will be contributed in the future. Thus, we conjecture that capabilities are essential for sustained growth. Although there are many reasons for high growth, such as the business context, a vibrant community, or a huge market demand (especially for mobile phones), we hypothesize that:

Hypothesis 6. *Capability-based dependencies sustain the growth of an ecosystem.*

However, although capabilities foster decentralized variability, they rely on a stable and centralized vocabulary. Yet, investigating dynamics of vocabulary creation and reuse in an ecosystem, remains an interesting research question. Understanding this aspect could enhance facilities to support app interactions and reuse.

9. Threats to Validity

9.1. External Validity

To enhance external validity, we selected large, substantial cases of ecosystems covering fairly diverse domains, technologies, and organizational structures. Although representativeness of subjects is not required for theory building from cases, our selection can be seen as theoretic sampling [61]. Still, smaller ecosystems controlled by specialized companies in a narrow market segment (niche players) might have different characteristics not covered by our conceptual framework. We mitigate this threat by using an exploratory research method: instead of testing hypotheses, we record phenomena and carefully develop hypotheses.

We analyzed the subjects as they are. A limitation is that we did not systematically elicit goals and requirements that led to the choice of mechanisms. Although the identified mechanisms are clearly driven by requirements of the domain (c.f., Hypothesis 4), performing a goal and requirements analysis, and linking the requirements to characteristics of the mechanisms, would be valuable future work, but would require interviewing platform suppliers.

For the quantitative analysis, we rely on subsets of the ecosystem, which might not be representative. Thus, we considered the main distribution channels. Tables 5 and 2 show that we covered significant parts of the ecosystems.

9.2. Internal Validity.

To enhance internal validity, we limited our data sources to reliable documents, freely available source code, and tools. Our observations were triangulated from these sources. This strategy aimed at reducing bias due to inappropriate sources. We also assigned one author as an expert to each subject, who deeply analyzed it. We cross-checked the within-case write-ups (further field notes are available from us on request). This strategy aimed at reducing interpretation bias of mechanism characteristics. To develop hypotheses, we followed a systematic approach. According to the conceptual framework, we performed a cross-case analysis, which identified major differences. We propose explanations for these, and formulate the most significant relationships as hypotheses. We provide datasets, details on data sources, additional diagrams, and our analysis tools in an online repository. This strategy makes our calculations transparent, to mitigate bias from incorrect statistics. In fact, some numbers are estimated using interpolations and safe assumptions (lower bounds).

Recall that all ecosystems except Android declare dependencies. It is not clear whether our extracted dependencies for Android are comparable to declared dependencies. In fact, it is subject of ongoing research, whether actual and declared dependencies are generally comparable or not. Furthermore, we could not analyze reverse directions of direct and capability-based dependencies in Android, given the intricate intent matching algorithm which we could not emulate. Therefore, we avoid comparing dependency numbers for Android to other systems. Still, all numbers indicate scalability requirements for tools. In this sense (algorithmic hardness), they are useful standalone and, to a large extent, comparable.

9.3. Construct Validity

In the qualitative analysis, the selection of dimensions for the conceptual framework was driven by our subjects. For different ecosystems, we could have potentially produced a different taxonomy. To mitigate the risk of incompleteness, we performed a detailed domain analysis of this space, covering the last three areas of the BAPO taxonomy [62]. The results were rich enough to create a consistent conceptual framework describing the five subjects by means of the dimensions.

Eclipse has recently introduced the generic provisioning system p2 [63], which abstracts over OSGi bundles and replaces Eclipse's bundle-based installer tool. However, our qualitative and quantitative analysis only analyzes bundles. This limitation is acceptable, since bundles are the primary building blocks of the main platform, and since the main free market repository (Eclipse Marketplace) still refers to bundles in individual update sites, which we mined.

To analyze dependencies, we constructed measures for specific types of dependencies, which was also driven by our subjects. However, dependency types might not be well enough defined to precisely construct platform-specific analyses. We mitigate this threat by creating a dependency metamodel in a bottom-up way, providing further details in an Appendix [14]. But the expressive facilities required some abstractions. The Debian and Eclipse dependency statistics disregard version numbers. This limitation is acceptable—both datasets have no packages in different versions. For Eclipse, we also disregard services. This limitation has (if any) only minor impact, as services are rarely used by the Eclipse community [64]. Future work might investigate these service-based interactions, but since they are codified in the program logic, that would likely require a static analysis as complex as our Android analysis.

10. Related Work

In the field of software ecosystems, our work contributes to research on theory building, architectural openness, variability mechanisms, and dependencies.

10.1. Software Ecosystem Theory Building

Barbosa et al. [15] review publications on software ecosystems using a mapping study. They identify ecosystem modeling, ecosystem architectures, licensing, and software evolution as main challenges. In addition to many theoretical works, the study discovers ten qualitative case studies identifying characteristics of cosystems. Our conceptual framework of technical aspects contributes to these case studies. Hanssen et al. [13] review literature on software ecosystems. They target works about theory building around the "rather vague and diverse" concept of a software ecosystem. They emphasize the lack of unified terminology and well-defined concepts that characterize ecosystems, and request theory-building research. Our work contributes carefully developed hypotheses about technical mechanisms. Jansen et al. [12] present a research agenda for software ecosystems, proposing to study ecosystems such as MySQL/PHP, Microsoft Windows, and iPhone apps. We deliver on this agenda by investigating similar systems. They announce the characterization and modeling of ecosystems as a main challenge, which we address with empirical data.

10.2. Architectural Openness

Anvaari et al. [9] have studied architectural openness of ecosystem platforms before. They analyze the five mobile application platforms Android, Symbian, Windows Mobile, Blackberry, and iPhone using a literature review and developer interviews. They discuss openness strategies, corresponding platform architectures, and their variability mechanisms ("extension mechanisms"). Their results also indicate that openness is a spectrum, and that non-technical aspects have a significant impact. For instance, although Symbian and Android are technically most open, in practice, other issues hinder contributions, such as a strict filtering of outside contributions to the main platform or licensing issues. We complement their work with a qualitative and quantitative analysis of variability mechanisms. Interestingly, the authors discuss a three-layered architecture of mobile application platforms (apps, middleware, kernel). From that perspective, our work investigates each layer separately, as our subjects cover each of them:

Android (app layer), Debian (middleware layer), and Linux (kernel layer).

10.3. Variability Mechanisms

Variability mechanisms have been discussed from various perspectives. Schmid et al. [65] discuss variability ("customization") mechanisms in service platforms, based on a literature review and an industry partner's yard management system (YMS). They describe various forms of variability occurring in the platform, and identify static and dynamic variability mechanisms suited for service-oriented platforms from the literature. They discuss their shortcomings and propose coarse-grained mechanisms applicable to the YMS. Their work is related, as service platforms can be the basis of ecosystems. However, in contrast to a literature review and theoretical discussion of mechanisms, we empirically study large-scale platforms, two of which (Eclipse, Android) have a service-oriented character.

Other works provide transformations between variability models and manifests. Cosmo et al. [66] show how feature models can be encoded as interdependent manifests. Galindo et al. [67] interpret Debian manifests as one feature model. In both works, the feature modeling languages are much simpler than the languages we investigated. Another comparative study of Eclipse and Debian manifests was done by Schmid [68]. He concludes that constraint declaration facilities of Debian and Eclipse manifests are comparable in their expressiveness to feature models, with minor limitations. He also identifies a number of concepts (e.g., versioning and information hiding) to handle variability in a distributed development structure.

We extend the latter three works. Our results challenge the practicality of transformations between real-world variability models and manifests, due to the diverse expressiveness we observed and Hypothesis 1 that such models are not applicable in a distributed setting.

10.4. Dependencies

Researchers have studied dependencies in ecosystems. Lungo et al. [69] recover dependencies between related software projects—the main entities in their notion of a software ecosystem. They assume that projects are linked together in some form, and consider dependencies originating from method calls and class references among the projects. They propose a metamodel and instantiate it by mining 211 Smalltalk projects. Robbes et al. [70] present early results on a study of API ripple effects (forced maintenance of code when a used API changes) in two ecosystems. They consider structural changes: addition, removal, and renaming of classes and methods. They observe that ripple effects can have a long lifecycle (up to four months), during which assets remain in an inconsistent state.

Both works consider ecosystems as collections of individual software projects, managed within their own source code repositories and having references to each other, not necessarily relying on a common run-time platform, but using common programming language mechanisms (e.g., class referencing, method calls). This view of ecosystems is different from ours. In fact, in our three open platforms, the relationship between basic units and projects is not obvious. While many basic units might be related to exactly one software development project, we believe this relationship is more diverse. Thus, their identified dependency structures are different; theirs and our work provide complementary views. Finally, we also contribute a technique to discover dependencies between Android apps, with a static analysis of Android (Dalvik) bytecode.

11. Conclusion

With our exploratory study of five successful ecosystems, we took one, but self-contained step towards building a theory. We explored the spectrum of mechanisms, built a conceptual framework, and used it to compare the subjects. We propose explanations for the differences and develop hypotheses with practical implications for project management, architecture, and tool support.

Conceptual Framework. The conceptual framework was instrumental to compare our subjects. Beyond being a helper construct, it also aims at understanding variability mechanisms, their characteristics and relationships. For instance, it relates variability models, manifests, and types of assets. It allows reasoning about ecosystems with a common terminology. Characteristics in the framework reflect important design decisions that platform providers have to make. Instantiations of the framework for each of the studied ecosystems provide insight into the concrete solutions to variability, and the organizational contexts in which they are used. The framework can be extended or refined by other researchers when conducting further studies of ecosystem technology. It can also be related to other frameworks addressing non-technical aspects.

Variability Mechanisms. We learn that variability models are effective in centralized variability management scenarios, and particularly for systems software. It is not clear whether they would be beneficial in a distributed variability management scenario. In all the studied ecosystems, we find rich dependency languages and variability descriptions comprising many direct and indirect (*capability-based*) dependencies. Indirect dependencies to abstract capabilities, as opposed to concrete units, are used intensively in highly growing ecosystems. Ecosystems with large free markets of assets available for end users offer much simpler dependency languages, respecting the less technical nature of the consumers of their product. On the other hand, they rely on more expressive capabilities, which should not just be labels, but DSLs with an extensive vocabulary.

Variability Encouragement. Recall that variability management in closed platforms, such as software product lines, aims at taming variability, to avoid diversity that has no business advantage. This objective is supported by activities such as variability modeling, scoping (controlling and restricting contributions), and maintaining variability information of basic units (unit parameters, dependencies, versioning). The involved variability mechanisms are rather heavyweight and require advanced technical skills, which hinders contributions. The mechanisms used by open platforms target a different set of activities. Among others, these comprise maintaining capabilities and common vocabularies, establishing uniform distribution channels, and lowering the entry barriers for contributions while assuring their quality using technical mechanisms. The identified mechanism of very different nature in open ecosystem platforms calls for recognition of a new discipline in variability research: *Variability Encouragement*. Analyzing the activities behind it and relating them to known software engineering processes and practices is an agenda for future research.

Acknowledgements

We would like to thank Sven Apel, Bram Adams, Christian Kästner, and Klaus Schmid for discussions on earlier versions of this article, as well as the anonymous reviewers for constructive feedback. This work was supported by the German National Academic Foundation, the German Research Foundation (LO 1719/3-1), the German Federal Ministry of Education and Research (01IS10033D), and the Ontario Research Fund.

References

[1] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, O. Spinczyk, Is the Linux Kernel a Software Product Line?, in: SPLC-OSSPL, 2007.

[2] T. Berger, S. She, R. Lotufo, A. Wasowski, K. Czarnecki, A study of variability models and languages in the systems software domain, IEEE Transactions on Software Engineering 39 (12) (2013) 1611–1640.

[3] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[4] J. Bosch, From software product lines to software ecosystems, in: SPLC, 2009.

[5] J. Corbet, G. Kroah-Hartman, A. McPherson, Linux kernel development, http://go.linuxfoundation.org/who-writes-linux-2012 (2012).

[6] C. Burkard, T. Widjaja, P. Buxmann, Software ecosystems, Wirtschaftsinformatik 54.

[7] J. D. McGregor, Ecosystems continued, Journal of Object Technology 8 (7).

[8] J. van Gurp, C. Prehofer, J. Bosch, Comparing practices for reuse in integration-oriented software product lines and large open source software projects, Software: Practice and Experience 40 (4) (2010) 285–312.

[9] M. Anvaari, S. Jansen, Evaluating architectural openness in mobile software platforms, in: ECSA, 2010.

[10] J. V. Gurp, J. Bosch, M. Svahnberg, On the notion of variability in software product lines, in: WICSA, 2001.

[11] M. Svahnberg, J. van Gurp, J. Bosch, A taxonomy of variability realization techniques, Software: Practice and Experience 35 (8) (2005) 705–754.

[12] S. Jansen, A. Finkelstein, S. Brinkkemper, A sense of community: A research agenda for software ecosystems, in: ICSE, 2009.

[13] G. K. Hanssen, T. Dybå, Theoretical foundations of software ecosystems, in: IWSECO, 2012.

[14] Online Appendix, http://bitbucket.org/tberger/ecosystem_mining.

[15] O. Barbosa, C. Alves, A systematic mapping study on software ecosystems, in: IWSECO, 2011.

[16] D. G. Messerschmitt, C. Szyperski, Software Ecosystem: Understanding an Indispensable Technology and Industry, MIT Press, 2003.

[17] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 2002.

[18] IT Radar, Software ecosystems - interview with Slinger Jansen, http://www.it-radar.org/serendipity/uploads/transkripte/SECO-Transcript_I.pdf (2012).

[19] C. Seidl, U. Assmann, Towards modeling and analyzing variability in evolving software ecosystems, in: VaMoS, 2013.

[20] G. K. Hanssen, A longitudinal case study of an emerging software ecosystem: Implications for practice and theory, J. Syst. Softw. 85 (7) (2012) 1455–1466.

[21] M. Svahnberg, J. van Gurp, J. Bosch, On the notion of variability in software product lines, Tech. Rep. Research Report No. 02/01, Blekinge Institute of Technology (2001).

[22] D. Muthig, T. Patzke, Generic implementation of product line components, in: NetObjectDays, 2002.

[23] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Tech. Rep. SEI-90-TR-21, CMU-SEI (1990).

[24] K. Schmid, R. Rabiser, P. Grünbacher, A comparison of decision modeling approaches in product lines, in: VaMoS, 2011.

[25] A. Hubaux, Y. Xiong, K. Czarnecki, A user survey of configuration challenges in Linux and eCos, in: VaMoS, 2012.

[26] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian, Selecting empirical methods for software engineering research, in: Guide to Advanced Empirical Software Engineering, Springer, 2008.

[27] M. Jørgensen, D. Sjøberg, Generalization and theory-building in software engineering research, in: EASE, 2004.

[28] K. M. Eisenhardt, Building theories from case study research, The Academy of Management Review 14 (4) (1989) 532–550.

[29] K. Czarnecki, U. W. Eisenecker, Generative Programming, Addison-Wesley, 2000.

[30] Debian policy, http://debian.org/doc/debian-policy, accessed 06/2013.

[31] Eclipse Development Process, http://eclipse.org/projects/dev_process/development_process_2010.pdf, accessed 06/2013.

[32] T. Berger, Variability modeling in the real, Ph.D. thesis, Faculty of Mathematics and Computer Science, University of Leipzig, available at http://nbn-resolving.de/urn:nbn:de:bsz:15-qucosa-113623 (May 2013).

[33] eCos and RedBoot based products showcase, http://ecoscentric.com/ecos/examples.shtml, accessed 06/2013.

[34] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, D. M. German, Macro-level software evolution: a case study of a large software compilation, Empirical Software Engineering 14 (3) (2009) 262–285.

[35] J. D. McGregor, J. Y. Monteith, Eclipse: An ecosystem case study, SPLC'12 tutorial on Supporting Strategic Software Engineering Decision Making through Ecosystems (2012).

[36] M. Milinkovich, Eclipse: The open innovation network, Presentation at Open Source Meets Business. Slides available at http://www.heise.de/events/2007/open_source_meets_business/keynotes/vortrag117.pdf (2007).

[37] eCos, http://ecos.sourceware.org/, accessed 06/2013.

[38] J. Corbet, G. Kroah-Hartman, A. McPherson, Linux kernel development, https://www.linuxfoundation.org/sites/main/files/lf_linux_kernel_development_2010.pdf (2010).

[39] M. Krafft, The Debian System, Open Source Press, 2005.

[40] Eclipse marketplace, http://marketplace.eclipse.org, accessed 06/2013.

[41] Yoxos on Demand, http://ondemand.yoxos.com, accessed 06/2013.

[42] Android Open Source Project – People and Roles, http://source.android.com/source/roles.html, accessed 06/2013.

[43] I. Herraiz, A. E. Hassan, Beyond lines of code: Do we need more complexity metrics?, in: A. Oram, G. Wilson (Eds.), Making Software: What Really Works, and Why We Believe It, O'Reilly Media, 2010, pp. 125–141.

[44] Unofficial Debian Repositories, http://apt-get.org, accessed 06/2013.

[45] Number of available Android applications, http://appbrain.com/stats/number-of-android-apps, accessed 06/2013.

[46] T. Berger, S. She, K. Czarnecki, A. Wąsowski, Feature-to-Code

mapping in two large product lines, in: SPLC, 2010.
[47] S. Nadi, R. Holt, The linux kernel: A case study of build system variability, Journal of Software: Evolution and Process.
[48] T. Berger, S. She, Formal semantics of the CDL language, http://informatik.uni-leipzig.de/~berger/cdl_semantics.pdf (2010).
[49] S. She, T. Berger, Formal semantics of the Kconfig language, http://eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf (2010).
[50] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, R. Treinen, Managing the complexity of large free and open source package-based software distributions, in: ASE, 2006.
[51] J. McAffer, P. VanderLei, S. Archer, OSGi and Equinox: Creating Highly Modular Java Systems, 1st Edition, Addison-Wesley Professional, 2010.
[52] Android Devolper Guide – App Manifest, http://developer.android.com/guide/topics/manifest/manifest-intro.html, accessed 01/2014.
[53] T. Thüm, C. Kästner, S. Erdweg, N. Siegmund, Abstract features in feature modeling, in: SPLC, 2011.
[54] Google Play, http://play.google.com, accessed 06/2013.
[55] E. Zini, A cute introduction to debtags, in: 5th annual Debian Conference, 2005.
[56] Some development model notes, http://lwn.net/Articles/108484, accessed 06/2013.
[57] R. Lotufo, S. She, T. Berger, K. Czarnecki, A. Wąsowski, Evolution of the Linux kernel variability model, in: SPLC, 2010.
[58] Android Devoloper Guide – Intents and Intent Filters, http://developer.android.com/guide/components/intents-filters.html, accessed 06/2013.
[59] OpenIntents, http://openintents.org, accessed 06/2013.
[60] M. Mendonca, A. Wąsowski, K. Czarnecki, Sat-based analysis of feature models is easy, in: SPLC, 2009.
[61] K. M. Eisenhardt, M. E. Graebner, Theory building from cases: Opportunities and challenges., Academy of management journal 50 (1) (2007) 25–32.
[62] H. Obbink, J. Müller, P. America, R. van Ommering, G. Muller, W. van der Sterren, J. Wijnstra, COPA: a component-oriented platform architecting method for families of software-intensive electronic products, Tutorial for SPLC.
[63] D. Le Berre, P. Rapicault, Dependency Management for the Eclipse Ecosystem: Eclipse p2, Metadata and Resolution, in: IWOCE, 2009.
[64] Refcard Equinox & OSGi, http://cdn.dzone.com/sites/all/files/refcardz/rc037-010d-equinox.pdf, accessed 12/2013.
[65] K. Schmid, H. Eichelberger, C. Kröher, Domain-oriented customization of service platforms: Combining product line engineering and service-oriented computing, Journal of Universal Computer Science 19 (2) (2013) 233–253.
[66] R. D. Cosmo, S. Zacchiroli, Feature diagrams as package dependencies, in: SPLC, 2010.
[67] J. A. Galindo, D. Benavides, S. Segura, Debian packages repositories as Software Product Line models., in: ACoTA, 2010.
[68] K. Schmid, Variability modeling for distributed development - a comparison with established practice, in: SPLC, 2010.
[69] M. Lungu, R. Robbes, M. Lanza, Recovering inter-project dependencies in software ecosystems, in: ASE, 2010.
[70] R. Robbes, M. Lungu, A study of ripple effects in software ecosystems, in: ICSE, 2011.