Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems

Walt Scacchi

Institute for Software Research. University of California, Irvine USA

Thomas A. Alspaugh

Computer Science Dept., Georgetown University, Washington, DC, USA

Abstract

The role of software ecosystems in the development and evolution of heterogeneously-licensed open architecture systems has received insufficient consideration. Such systems are composed of components potentially under two or more licenses, open source or proprietary or both, in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring. The software licenses of the components both facilitate and constrain the system's ecosystem and its evolution, and the licenses' rights and obligations are crucial in producing an acceptable system. Consequently, software component licenses and the architectural composition of a system determine the software ecosystem niche where a systems lies. Understanding and describing software ecosystem niches is a key contribution of this work. A case study of an open architecture software system that articulates different niches is employed to this end. We examine how the architecture and software component licenses of a composed system at design-time, build-time, and run-time help determine the system's software ecosystem niche and provide insight and guidance for identifying and selecting potential evolutionary paths of system, architecture, and niches.

Keywords: Software architecture, software ecosystems, software licenses, open

Email addresses: wscacchi@ics.uci.edu (Walt Scacchi), alspaugh@cs.georgetown.edu (Thomas A. Alspaugh)

source software, software evolution, case study.

---

1. Introduction

A substantial number of development organizations are adopting a strategy in which a software-intensive system (one in which software plays a crucial role) is developed with an *open architecture* (OA) [29], whose components may be open source software (OSS) or proprietary with open application programming interfaces (APIs). Such systems evolve not only through the evolution of their individual components, but also through replacement of one component by another, possibly from a different producer or under a different license. With this approach, another organization often comes between software component producers and system consumers. These organizations take on the role of system architect or integrator, either as independent software vendors, government contractors, system integration consultants, or in-house system integrators. In turn, such an integrator designs a system architecture that can be composed of components largely produced elsewhere, interconnected through interfaces accommodating use of dynamic links, intra- or inter-application scripts, communication protocols, software buses, databases/repositories, plug-ins, libraries or software shims as necessary to achieve the desired result. An OA development process results in an ecosystem in which the integrator is influenced from one direction by the goals, interfaces, license choices, and release cycles of the software component producers, and from another direction by the needs of the system's consumers. As a result the software components are reused more widely, and the resulting OA systems can achieve reuse benefits such as reduced costs, increased reliability, and potentially increased agility in evolving to meet changing needs. An emerging challenge is to realize the benefits of this approach when the individual components are *heterogeneously licensed* [2, 17, 33], each potentially with a different license, rather than a single OSS license as in uniformly-licensed OSS projects or a single proprietary license as in proprietary development.

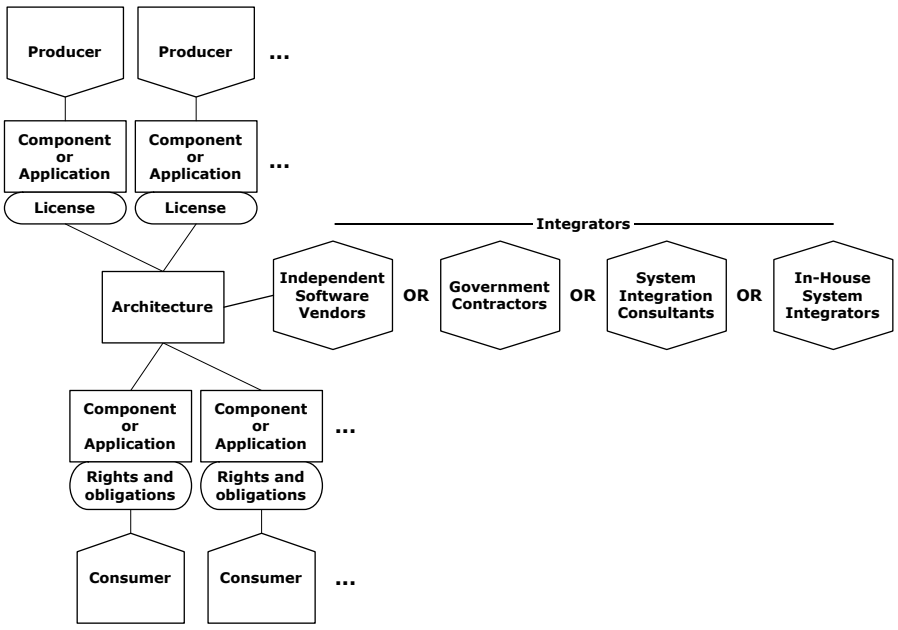

Figure 1: Schema for software supply networks

This challenge is inevitably entwined with the software ecosystems that arise for OA systems (Figure 1). We find that an OA software ecosystem involves organizations and individuals producing, composing, and consuming components that articulate software supply networks from producers to consumers, but also

- the OA of the system(s) in question,
- the open interfaces met by the components,
- the degree of coupling in the evolution of related components, and
- the rights and obligations resulting from the software licenses under which various components are released, that propagate from producers to consumers.

A motivating example of this approach is the Unity game development tool, produced by Unity Technologies [35]. Its license agreement, from which we quote below, lists eleven distinct licenses and indicates the tool is produced, apparently

using an OA approach, using at least 18 externally produced components or groups of components:

1. The Mono Class Library, Copyright 2005-2008 Novell, Inc.
2. The Mono Runtime Libraries, Copyright 2005-2008 Novell, Inc.
3. Boo, Copyright 2003-2008 Rodrigo B. Oliveira
4. UnityScript, Copyright 2005-2008 Rodrigo B. Oliveira
5. OpenAL cross platform audio library, Copyright 1999-2006 by authors.
6. PhysX physics library. Copyright 2003-2008 by Ageia Technologies, Inc.
7. libvorbis. Copyright (c) 2002-2007 Xiph.org Foundation
8. libtheora. Copyright (c) 2002-2007 Xiph.org Foundation
9. zlib general purpose compression library. Copyright (c) 1995-2005 Jean-loup Gailly and Mark Adler
10. libpng PNG reference library
11. jpeglib JPEG library. Copyright (C) 1991-1998, Thomas G. Lane.
12. Twilight Prophecy SDK, a multi-platform development system for virtual reality and multimedia. Copyright 1997-2003 Twilight 3D Finland Oy Ltd
13. dynamic_bitset, Copyright Chuck Allison and Jeremy Siek 2001-2002.
14. The Mono C# Compiler and Tools, Copyright 2005-2008 Novell, Inc.
15. libcurl. Copyright (c) 1996-2008, Daniel Stenberg <daniel@haxx.se>.
16. PostgreSQL Database Management System
17. FreeType. Copyright (c) 2007 The FreeType Project (www.freetype.org).
18. NVIDIA Cg. Copyright (c) 2002-2008 NVIDIA Corp.

The software ecosystem for Unity3D as a standalone software package is delimited by the diverse set of software components listed above. However the architecture that integrates or composes these components is closed and thus unknown to a system consumer, as is the manner in which the different licenses associated with these components impose obligations or provide rights to consumers, or on the other components to which they are interconnected. Subsequently, we see that software ecosystems can be understood in part by examining relationships between architectural composition of software components that are subject to different licenses, and this necessitates access to the system's architecture composition. By examining the open architecture of a specific composed software system, it becomes possible to explicitly identify the software ecosystem niche in which the system is embedded.

A software ecosystem constitutes a software supply network that connects software producers to integrators to consumers through licensed components

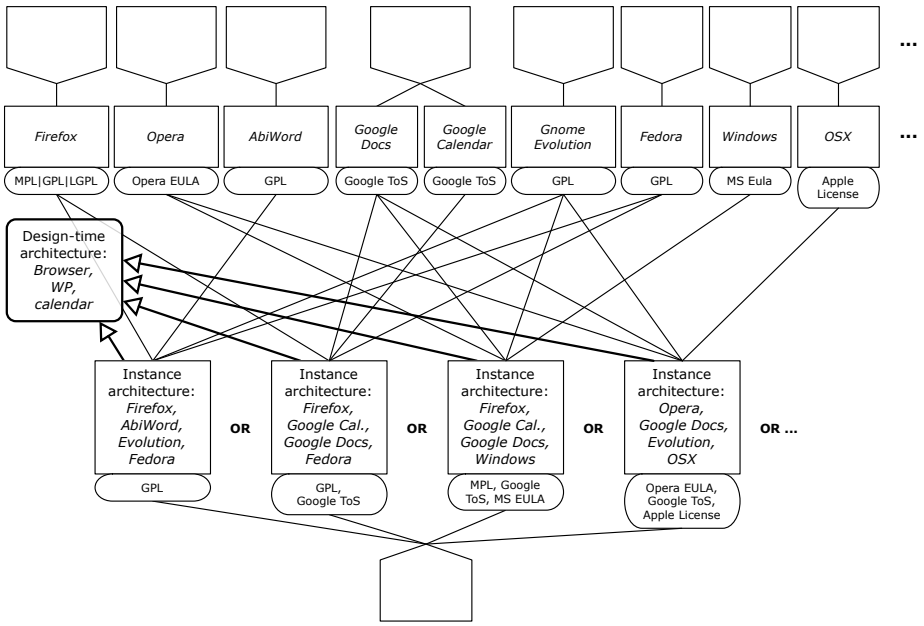

Figure 2: Ecosystem for three possible instantiations of a single design architecture

and composed systems. Figure 2 outlines the software ecosystem elements and relationships for an OA case study that we examine in this paper.

A *software ecosystem niche* articulates a specific software supply network that interconnects particular software producers, integrators, and consumers. A software system defined niche may lie within an existing single ecosystem, or one that may span a network of software producer ecosystems. A composed software system architecture helps determine the software ecosystem niche, since the architecture identifies the components, their licenses and producers, and thus the network of software ecosystems in which it participates. Such a niche also transmits license-borne obligations and access/usage rights passed from the participating software component producers through integrators and to system consumers. Thus, system architects or component integrators help determine in which software ecosystem niche a given instance architecture for the system participates.

As a software system evolves over time, as its components are updated or

changed, or their architectural interconnections are refactored, it is desirable to determine whether and how the system's ecosystem niche may have changed. A system may evolve because its consumers want to migrate to alternatives from different component producers, or choose components whose licenses are more/less desirable. Software system consumers may want to direct their system integrators to compose the system's architecture so as to move into or away from certain niches. Consequently, system integrators can update or modify system architectural choices at design-time, build-time (when components are compiled together into an executable), or run-time (when bindings to remote executable services are instantiated) to move a software system from one niche to another. Thus, understanding how software ecosystem niches emerge is a useful concept that links software engineering concerns for software architecture, system integration/composition, and software evolution to organizational and supply network relationships between software component producers, integrators and system consumers.

To help explain how OA systems articulate software ecosystem niches, we provide a software architecture case study for use in this paper. This system is intentionally simple for expository purposes. Its architectural design composes a web browser, word processor, calendaring and email applications, host platform operating system, and remote services as its components. Equivalent components from different OSS or proprietary software producers can be identified, where each alternative is subject to a different type of software license. For example, for Web browsers, we consider the Firefox browser from the Mozilla Foundation, which comes with a choice of OSS license (MPL, GPL, or LGPL), and the Opera browser from Opera Software, which comes with a proprietary software end-user license agreement (EULA). Similarly, for word processor, we consider the OSS AbiWord application (GPL) and Web-based Google Docs service (proprietary Terms of Service). The OA we describe covers a number of systems we have identified, built, and deployed in a university research laboratory. Example niches involving these components appear in Figures 3 and 4. We have also developed OA systems with more complex architectures that in-

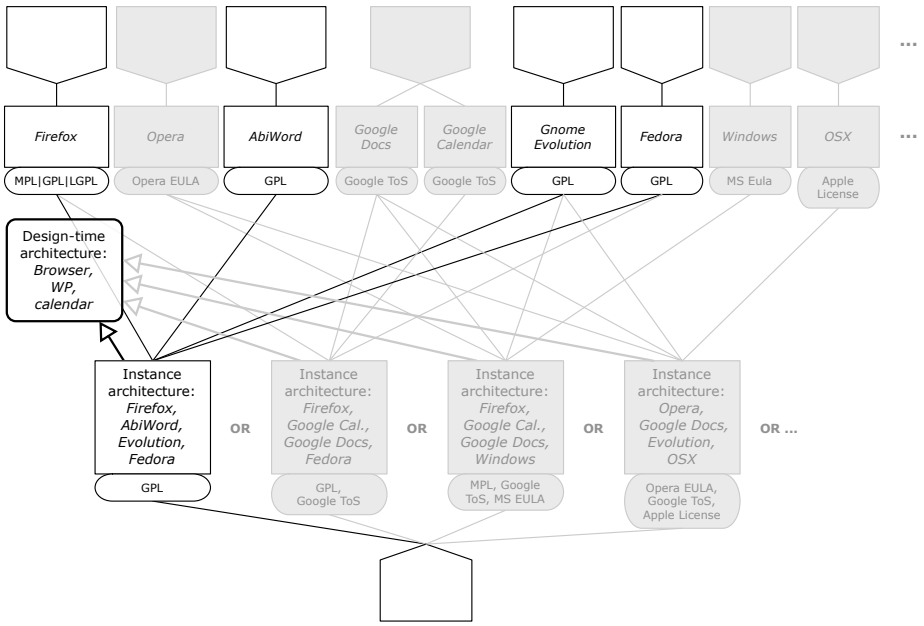

Figure 3: The ecosystem niche for one instance architecture

corporate components for content management systems (Drupal), wikis (MediaWiki), blogs (B2evolution), teleconferencing and media servers (Flash media server, Red5 media server), chat (BlaB! Lite), Web spiders and search engines (Nutch, Lucene, Sphider), relational database management systems (MySQL), and others. Furthermore, the OSS application stacks and infrastructure (platform) stacks found at BitNami.org/stacks (accessed 29 April 2010) could also be incorporated in OA systems, as could their proprietary counterparts. However, these more complex OAs still reflect the core architectural concepts and constructs, as well as software ecosystem relationships, that we present in our example case study in a simpler manner.

The software ecosystem niches for the case study system, or indeed any system, depend on which component implementations are used and the architecture in which they are combined and instantiated, as does the overall rights and obligations for the instantiated system. In addition, we build on previous work on heterogeneously-licensed systems [2, 17, 33] by examining how OA development

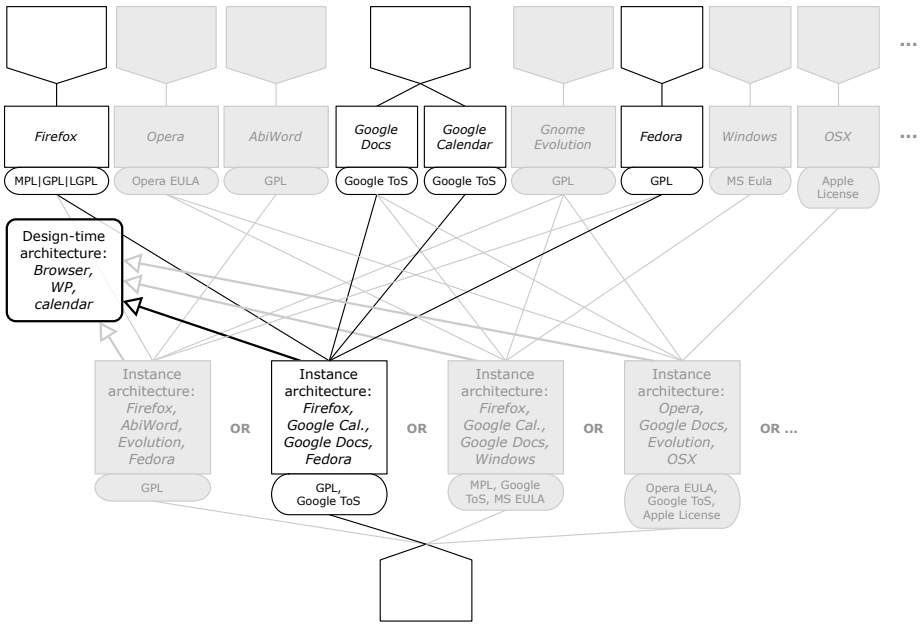

Figure 4: The ecosystem niche for a second instance architecture

affects and is affected by software ecosystems, and the role of component licenses in shaping OA software ecosystem niches.

Consequently, we focus our attention to understand the ecosystem of an open architecture software system such that:

- It must rest on a license structure of rights and obligations (Section 4), focusing on obligations that are enactable and testable[1].
- It must take account of the distinctions between the design-time, build-time, and distribution-time architectures (Section 3) and the rights and obligations that come into play for each of them.

[1]For example, many OSS licenses include an obligation to make a component's modified code public, and whether a specific version of the code is public at a specified Web address is both enactable (it can be put into practice) and testable. In contrast, the General Public License (GPL) v.3 provision "No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty" is not enactable in any obvious way, nor is it testable — how can one verify what others deem?

- It must distinguish the architectural constructs significant for software licenses, and embody their effects on rights and obligations (Section 3).
- It must define license architectures.
- It must account for alternative ways in which software systems, components, and licenses can evolve (Section 5), and
- It must provide an automated environment for creating and managing license architectures. We are developing a prototype that manages a license architecture as a view of the system architecture [3].

The remainder of this paper is organized as follows. Section 2 places this work in the context of related research. Section 3 discusses open architecture, and the influence of software licenses is discussed in Section 4. Section 5 addresses evolution of software ecosystems. Section 6 discusses some implications that follow from this study, and Section 7 concludes the paper.

2. Related Research

The study of software ecosystems is emerging as an exciting new area of systematic investigation and conceptual development within software engineering. Understanding the many possible roles that software ecosystems can play in shaping software engineering practice is gaining more attention since the concept first appeared [23]. Bosch [7] builds a conceptual lineage from software product line (SPL) concepts and practices [6, 13] to software ecosystems. SPLs focus on the centralized development of families of related systems from reusable components hosted on a common platform with an intra-organizational base, with the resulting systems either intended for in-house use or commercial deployments. Software ecosystems then are seen to extend this practice to systems hosted on an inter-organizational base, which may resemble development approaches conceived for virtual enterprises for software development [26]. Producers of commercial software applications or packages thus need to adapt their development strategy and business model to one focused on coordinating and

guiding decentralized software development of its products and enhancements (e.g., plug-in components).

Along with other colleagues [8, 12, 36], Bosch identifies alternative ways to connect reusable software components through integration and tight coupling found in SPLs, or via loose coupling using glue code, scripting or other late binding composition schemes in ecosystems or other decentralized enterprises [26, 27], as a key facet that can enable software producers to build systems from diverse sources.

Jansen and colleagues [18, 19] draw attention to their observation that software ecosystems (a) embed software supply networks that span multiple organizations, and (b) are embedded within a network of intersecting or overlapping software ecosystems that span the world of software engineering practice. Scacchi [32] for example, identifies that the world of open source software (OSS) development is a software ecosystem different from those of commercial software producers, and its supply networks are articulated within a network of FOSS development projects. Networks of OSS ecosystems have also begun to appear around very large OSS projects for Web browsers, Web servers, word processors, and others, as well as related application development environments like NetBeans and Eclipse, and these networks have become part of global information infrastructures [20].

OSS ecosystems also exhibit strong relationships between the ongoing evolution of OSS systems and their developer/user communities, such that the success of one co-depends on the success of the other [32]. Ven and Mannaert discuss the challenges independent software vendors face in combining OSS and proprietary components, with emphasis on how OSS components evolve and are maintained in this context [37].

Boucharas and colleagues [9] then draw attention to the need to more systematically and formally model the contours of software supply networks, ecosystems, and networks of ecosystems. Such a formal modeling base may then help in systematically reasoning about what kinds of relationships or strategies may arise within a software ecosystem. For example, Kuehnel [21] exam-

ines how Microsoft's software ecosystem developed around in operating systems (MS Windows) and key applications (e.g., MS Office) may be transforming from "predator" to "prey" in its effort to control the expansion of its markets to accommodate OSS (as the extant prey) that eschew closed source software with proprietary software licenses.

Our work in this area builds on these efforts in the following ways. First, we share the view of a need for examining software ecosystems, but we start from software system architectures that can be formally modeled and analyzed with automated tool support [6, 34]. Explicit modeling of software architectures enables the ability to view and analyze them at design-time, build-time, or deployment/run-time. Software architectures also serve as a mechanism for coordinating decentralized software development across multi-site projects [30]. Similarly, explicit models allow for the specification of system architectures using either proprietary software components with open APIs, OSS components, or combinations thereof, thereby realizing open architecture (OA) systems [33]. We then find value in attributing open architecture components with their (intellectual property) licenses [3], since software licenses are an expression of contractual/social obligations that software consumers must fulfill in order to realize the rights to use the software in specified allowable manners, as determined by the software's producers.

3. Open Architectures

Open architecture (OA) software is a customization technique introduced by Oreizy [29] that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. Increasingly more software-intensive systems are developed using an OA strategy, not only with OSS components but also proprietary components with open APIs (e.g. [35]). Using this approach can lower development costs and increase reliability and function [33]. Composing a system with heterogeneously-licensed components, however, increases the likelihood of conflicts, liabilities, and no-rights stemming from incompatible licenses. Thus, in our work we define an

OA system as a *software system consisting of components that are either open source or proprietary with open API, whose overall system rights at a minimum allow its use and redistribution, in full or in part.*

It may appear that using a system architecture that incorporate OSS components and uses open APIs will result in an OA system. But not all such architectures will produce an OA, since the (possibly empty) set of available license rights for an OA system depends on: (a) how and why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the licenses of different OSS components encumber all or part of a software system's architecture into which they are integrated [1, 33].

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed [5].

Software source code components—These can be either (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code such as C shell scripts, or (d) intra-application script code, as for creating Rich Internet Applications using domain-specific languages such as XUL for the Firefox Web browser [15] or "mashups" [25]. Their source code is available and they can be rebuilt. Each may have its own distinct license, though often script code that merely connects programs and data flows does not, unless the code is substantial, reusable, or proprietary.

Executable components—These components are in binary form, and the source code may not be open for access, review, modification, or possible redistribution [31]. If proprietary, they often cannot be redistributed, and so such components will be present in the design- and run-time architectures but not in the distribution-time architecture.

Software services—An appropriate software service can replace a source code or executable component.

Application programming interfaces/APIs—Availability of externally visible and accessible APIs is the minimum requirement for an "open system" [24]. Open APIs are not and cannot be licensed, but they can limit the propa-

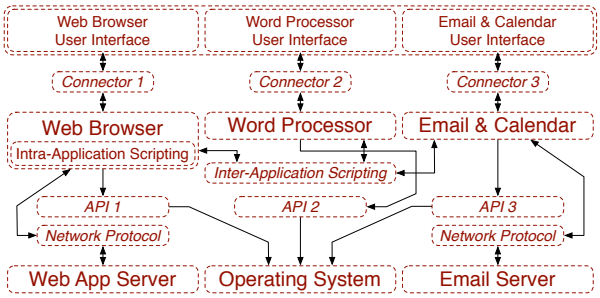

Figure 5: A design-time architecture

gation of license obligations.

Software connectors—Software whose intended purpose is to provide a standard or reusable way of communication through common interfaces, e.g. High Level Architecture [22], CORBA, MS .NET, Enterprise Java Beans, and GNU Lesser General Public License (LGPL) libraries. Connectors can also limit the propagation of license obligations.

Methods of composition—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. Methods of composition affect license obligation propagation, with different methods affecting different licenses.

Configured system or subsystem architectures—These are software systems that are used as atomic components of a larger system, and whose internal architecture may comprise components with different licenses, affecting the overall system license. To minimize license interaction, a configured system or sub-architecture may be surrounded by what we term a *license firewall*, namely a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of reciprocal obligations.

With these architectural elements, we can create an design-time or reference architecture for a system that conforms to the software supply network shown in Figure 2. This design-time architecture appears in Figure 5; note that it only specifies components by type rather than by producer, meaning the choice of producer component remains unbound at this point.

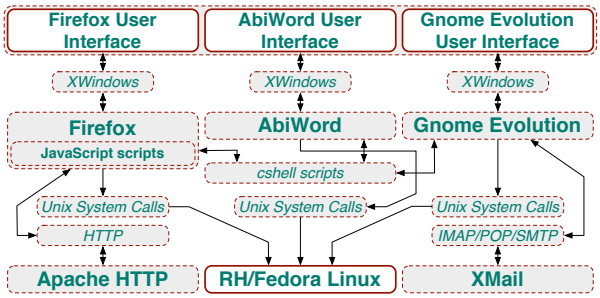

Figure 6: A build-time architecture

Then in Figure 6, we create a build-time rendering of this architectural design by selecting specific components from designated software producers. The gray boxes correspond to components and connectors not visible in the run-time instantiation of the system in Figure 7.

Figures 7, 8, and 9 display alternative run-time instantiations of the design-time architecture of Figure 5. The architectural run-time instance in Figure 7 corresponds to the software ecosystem niche shown in Figure 3; Figure 8 corresponds to the niche in Figure 4; and Figure 9 designates yet another niche different from the previous two.

Each component selection implies acceptance of the license obligations and rights that the producer seeks to transmits to the components consumers. However in an OA design development, component interconnections may be used to intentionally or unintentionally propagate these obligations onto other components whose licenses may conflict with them or fail to match [3, 17]; the system integrator can decide to insert software shims using scripts, dynamic links to remote services, data communication protocols, or libraries to mitigate or firewall the extent to which a component's license obligations propagate. This style of build-time composition can be used to accommodate a system's consumers' policy for choosing systems that avoid certain licenses, or that isolate the license obligations of certain desirable components. It also allows system integrators and consumers to follow a "best of breed" policy in the selection of system components. Finally, if no license conflicts exist in the system, or if the integra-

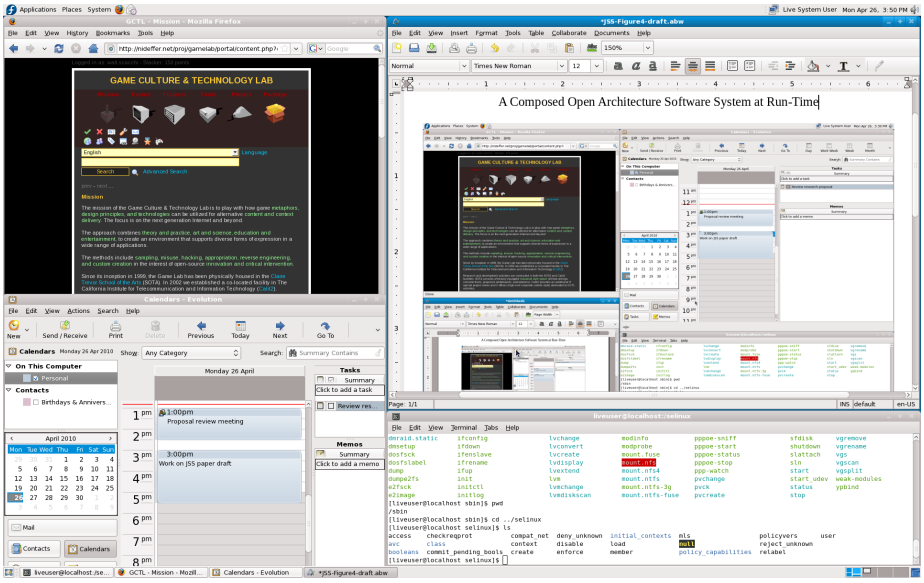

Figure 7: An instantiation at run-time (Firefox, AbiWord, Gome Evolution, Fedora) of the build-time architecture of Figure 6 that determines the ecosystem niche of Figure 3

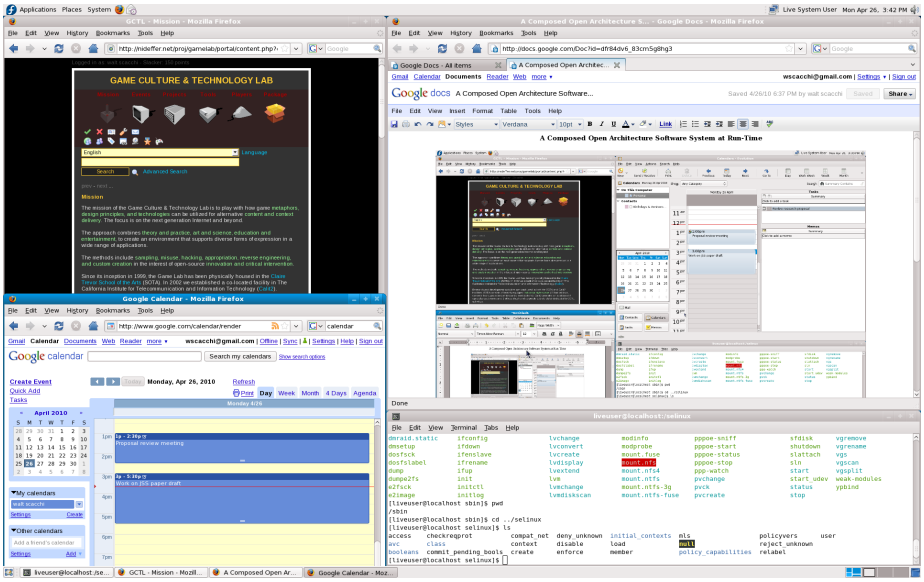

Figure 8: A second instantiation at run-time (Firefox, Google Docs and Calendar, Fedora) determining the ecosystem niche of Figure 4

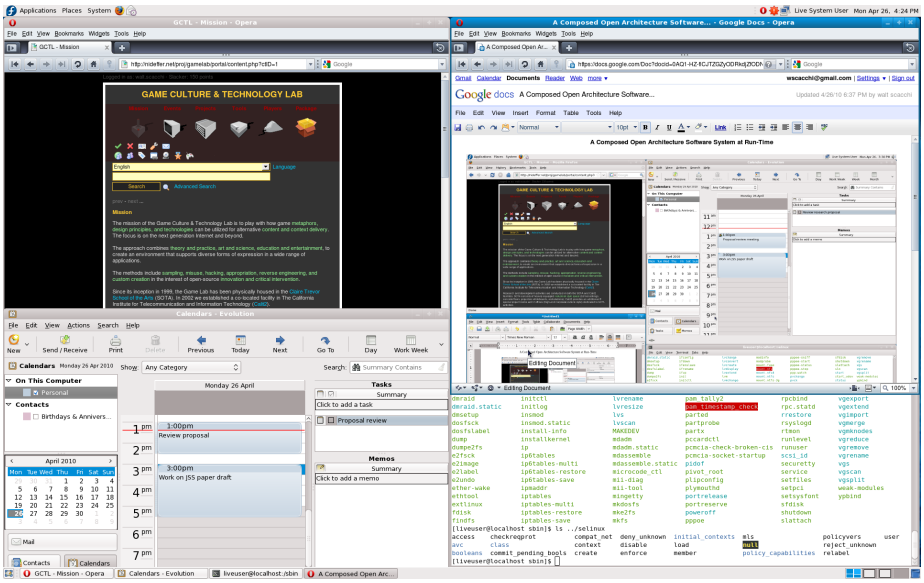

Figure 9: A third instantiation at run-time (Opera, Google Docs, Gnome Evolution, Fedora) determining yet another niche conforming to the software supply network of Figure 2

tor and system consumer are satisfied with the component and license choices made, then the compositional bindings may simply be set in the most efficient way available. This realizes a policy for accepting only components and licenses whose obligations and rights are acceptable to the system consumers.

4. Software Licenses

Traditional proprietary licenses allow a company to retain control of software it produces, and restrict the access and rights that outsiders can have. OSS licenses, on the other hand, are designed to encourage sharing and reuse of software, and grant access and as many rights as possible. OSS licenses are classified as *academic* or *reciprocal*. Academic OSS licenses such as the Berkeley Software Distribution (BSD) license, the Massachusetts Institute of Technology license, the Apache Software License, and the Artistic License, grant nearly all rights to components and their source code, and impose few obligations. Anyone can use the software, create derivative works from it, or include it in proprietary projects. Typical academic obligations are simply to not remove the copyright

and license notices.

Reciprocal OSS licenses take a more active stance towards sharing and reusing software by imposing the obligation that reciprocally-licensed software not be combined (for various definitions of "combined") with any software that is not in turn also released under the reciprocal license. Those for which most or all ways of combining software propagate reciprocal obligations are termed *strongly reciprocal*. Examples are the GPL and the Affero GPL (AGPL). The purpose of the AGPL is to prevent a GPL software component from being integrated into an OA system as a remote server, or from being wrapped with shims to inhibit its ability to propagate the GPL obligations and rights. The purpose of these licenses is to ensure that software so licensed will maintain (and can propagate) the freedom to access, study, modify, and redistribute the software source code, which academic licenses do not. This in turn assures the access, use, and reusability of the source code for other software producers and system integrators. Those licenses for which only certain ways of combining software propagate reciprocal obligations are termed *weakly reciprocal*. Examples are the Lesser GPL (LGPL), Mozilla Public License (MPL), and Common Public License. The goals of reciprocal licensing are to increase the domain of OSS by encouraging developers to bring more components under its aegis, and to prevent improvements to OSS components from vanishing behind proprietary licenses.

Both proprietary and OSS licenses typically disclaim liability, assert no warranty is implied, and obligate licensees to not use the licensor's name or trademarks. Newer licenses often cover patent issues as well, either giving a restricted patent license or explicitly excluding patent rights. The Mozilla Disjunctive Tri-License licenses the core Mozilla components under any one of three licenses (MPL, GPL, or the GNU Lesser General Public License LGPL); OSS developers or OA system integrators can choose the one that best suits their needs for a particular project and component.

The Open Source Initiative (OSI) maintains a widely-respected definition of "open source" and gives its approval to licenses that meet it [28]. OSI maintains

and publishes a repository of approximately 70 approved OSS licenses which tend to vary in the terms and conditions of their declared obligations and rights. However, all these licenses tend to cluster into either a strongly reciprocal, weakly reciprocal, or minimally restrictive/academic license type.

Common practice has been for an OSS project to choose a single license under which all its products are released, and to require developers to contribute their work only under conditions compatible with that license. For example, the Apache Contributor License Agreement grants enough of each author's rights to the Apache Software Foundation for the foundation to license the resulting systems under the Apache Software License. This sort of rights regime, in which the rights to a system's components are homogeneously granted and the system has a single well-defined OSS license, was the norm in the early days of OSS and continues to be practiced.

We have developed an approach for expressing software licenses that is more formal and less ambiguous than natural language, and that allows us to calculate and identify conflicts arising from the rights and obligations of two or more component's licenses. Our approach is based on Hohfeld's classic group of eight fundamental jural relations [Hohfeld 1913], of which we use right, duty, no-right, and privilege (Figure 10). We start with a tuple <actor, operation, action, object> for expressing a right or obligation. The actor is the "licensee" for all the licenses we have examined. The operation is one of the following: "may", "must", "must not", or "need not", with "may" and "need not" expressing rights and "must" and "must not" expressing obligations. Because copyright rights are only available to entities who have been granted a sublicense, only the listed rights are available, and the absence of a right means that it is not available. The action is a verb or verb phrase describing what may, must, must not, or need not be done, with the object completing the description. A license may be expressed as a set of rights, with each right associated with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 11 sketches two rights from GPL 2.0, the first one with no obligations and the second with three corresponding obligations. Elsewhere, the details of this license specification

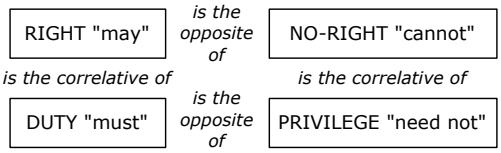

Figure 10: Hohfeld's four basic relations

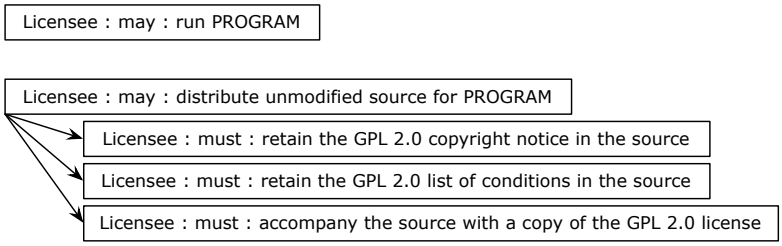

Figure 11: Tuples for some rights and obligations of the GPL 2.0 license

scheme, how it can be used to attribute software architectures to produce a system license architecture, and how it can be formalized into a semantic metamodel [3].

HLS designers have developed a number heuristics to guide architectural design while avoiding some license conflicts. First, it is possible to use a reciprocally-licensed component through a license firewall that limits the scope of reciprocal obligations. Rather than connecting conflicting components directly through static or other build-time links, the connection is made through a dynamic link, client-server protocol, license shim (such as an LGPL connector), or run-time plug-ins. A second approach used by a number of large organizations is simply to avoid using any components with reciprocal licenses. A third approach is to meet the license obligations (if that is possible) by for example retaining copyright and license notices in the source and publishing the source code. However, even using design heuristics such as these (and there are many), keeping track of license rights and obligations across components that are interconnected in complex OAs quickly becomes too cumbersome. Automated support is needed to manage the multi-component, multi-license complexity. Accordingly, we are developing an automated support capability as part of the

ArchStudio architecture design environment [14] that can analyze the addition of software license properties such as those shown in Figure 11 to the interfaces of software components in an OA system. For example, in Figure 12 we see a rendering of the OA system from our case study with the AbiWord word processing component highlighted. This component's APIs would be attributed with the GPL license obligations and rights in Figure 11, since AbiWord is licensed under GPL, as are other components like the Gnome Evolution calendaring and email application and also the Red Hat/Fedora Linux operating system platform. As the architectural interconnections shown in the model of Figure 12 indicate that none of these components covered by GPL are directly interconnected to another licensed component, their license obligations do not propagate or become "viral" in this architectural composition. Replacing any of these GPL components with non-GPL but still OSS components would not change the total set of obligations and rights on the system with this architecture; the system would remain OSS, but the software ecosystem niche in which it resides would shift to another niche. Once again, software licenses interact with software architectures and together they help determine which software ecosystem niche will embed an instantiated run-time version of the system.

5. Architecture, License, and Ecosystem Evolution

An OA system can evolve by a number of distinct mechanisms, some of which are common to all systems but others of which are a result of heterogeneous component licenses in a single system.

By component evolution— One or more components can evolve, altering the overall system's characteristics (for example, upgrading and replacing the Firefox Web browser from version 3.5 to 3.6). Such minor versions changes generally have no effect on system architecture.

By component replacement— One or more components may be replaced by others with modestly different functionality but similar interface, or with a different interface and the addition of shim code to make it match (for example, replacing the AbiWord word processor with either Open Office Writer or MS

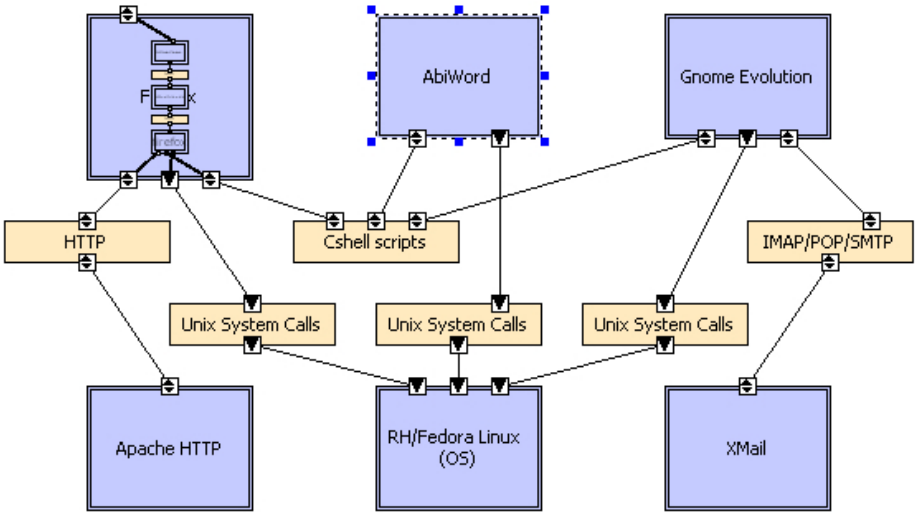

Figure 12: The model of the system architecture used in our case study as rendered in our automated tool, described elsewhere [2, 3]

Word). However, changes in the format or structure of component APIs may necessitate build-time and run-time updates to component connectors. Figure 13 shows some possible alternative system compositions that result from replacing components by others of the same type but with a different license.

By architecture evolution— The OA can evolve, using the same components but in a different configuration, altering the system characteristics. For example, as discussed in Section 4, revising or refactoring the configuration in which a component is connected can change how its license affects the rights and obligations for the overall system. This could arise when replacing word processing, calendaring, and email components and their connectors with Web-browser-based services such as Google Docs, Google Calendar, and Google Mail. The replacement would eliminate the legacy components and relocate the desired application functionality to operate within the Web browser component, resulting in what might be considered a simpler and easier-to-maintain system architecture, but one that is less open and now subject to a proprietary Terms of Service license. System consumer policy preferences for licenses, and subsequent participation in a different ecosystem niche, may thus mediate whether

such an alternative system architecture is desirable or not.

By component license evolution— The license under which a component is available may change, as for example when the license for the Mozilla core components was changed from the Mozilla Public License (MPL) to the current Mozilla Disjunctive Tri-License; or the component may be made available under a new version of the same license, as for example when the GNU General Public License (GPL) version 3 was released. The three architectures in Figure 13 that incorporate the Firefox Web browser show how its tri-license creates new evolutionary paths by offering different licensing options. These options and paths were not available previously with earlier versions of this component offered under only one or two license alternatives.

By a change to the desired rights or acceptable obligations— The OA system's integrator or consumers may desire additional license rights (for example the right to sublicense in addition to the right to distribute), or no longer desire specific rights; or the set of license obligations they find acceptable may change. In either case the OA system evolves, whether by changing components, evolving the architecture, or other means, to provide the desired rights within the scope of the acceptable obligations. For example, they may no longer be willing or able to provide the source code for components within the reciprocality scope of a GPL-licensed module. Figure 14 shows an array of choices among types of licenses for different types of components that appear in the OA case study example. Each choice determines the obligations that component producers can demand of their consumers in exchange for the access/usage rights they offer

The interdependence of producers, integrators, and consumers results in a coevolution of software systems and social networks within an OA ecosystem [32]. Closely-coupled components from different producers must evolve in parallel in order for each to provide its services, as evolution in one will typically require a matching evolution in the other. Producers may manage their evolution with a loose coordination among releases, for example as between the Gnome and Mozilla organizations. Each release of a producer component create a tension

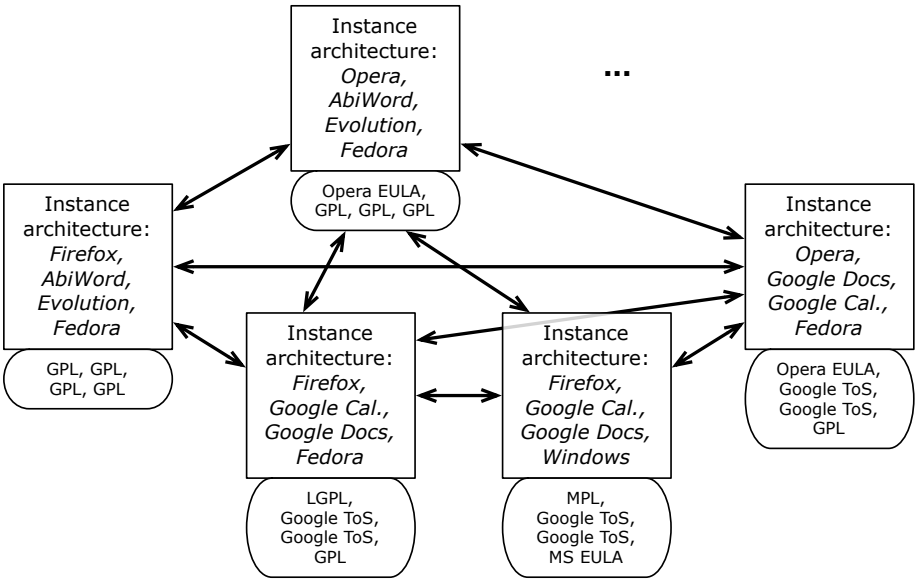

Figure 13: Possible evolutionary paths among a few instance architectures; some paths are impractical due to the changes in license obligations

	Browser	**Word processor**	**Calendar, email**	**Platform**
Proprietary	Opera (Opera EULA)	WordPerfect (Corel License)		Windows (MS EULA)
Strongly Reciprocal	Firefox (MPL or LGPL or GPL)	AbiWord (GPL)	Gnome Evolution (GPL)	Fedora (GPL)
Weakly Reciprocal or Academic		OpenOffice (LGPL)		FreeBSD (BSD variant)
Service		Google Docs (Google ToS)	Google Calendar (Google ToS)	

Figure 14: Some architecture choices and their license categories

through the ecosystem relationships with consumers and their releases of OA systems using those components, as integrators accommodate the choices of available, supported components with their own goals and needs. As discussed in our previous work [3], license rights and obligations are manifested at each component interface then mediated through the OA of the system to entail the rights and corresponding obligations for the system as a whole. As a result, integrators must frequently re-evaluate the OA system rights and obligations. In contrast to homogeneously-licensed systems, license change across versions is a characteristic of OA ecosystems, and architects of OA systems require tool support for managing the ongoing licensing changes.

6. Discussion

At least two topics merit discussion following from our approach to understanding of software ecosystems and ecosystem niches for OA systems: first, how might our results shed light on software systems whose architectures articulate a software product line; and second, what insights might we gain based on the results presented here on possible software license architectures for mobile computing ecosystems. Each is addressed in turn.

Software product lines (SPLs) rely on the development and use of explicit software architectures [6, 13]. However, the architecture of an SPL or software ecosystem does not necessarily require an OA—there is no need for it to be open. Thus, we are interested in discussing what happens when SPLs may conform to an OA, and to an OA that may be subject to heterogeneously licensed SPL components. Three considerations come to mind:

1. If the SPL is subject to a single homogeneous software license, which may often be the case when a single vendor or government contractor has developed the SPL, then the license may act to reinforce a vendor lock-in situation with its customers. One of the motivating factors for OA is the desire to avoid such lock-in, whether or not the SPL components have open or standards-compliant APIs. However, a single license simplifies determination of the software ecosystem in which these system is located.

2. If an OA system employs a reference architecture, then such a reference or design-time architecture effectively defines an SPL consisting of possible different system instantiations composed from similar components from different producers (e.g., different but equivalent Web browsers, word processors, calendaring and email applications). This can be seen in the design- time architecture depicted in Figure 5, the build-time architecture in Figure 6, and the instantiated run-time architectures in Figures 7, 8, and 9.
3. If the SPL is based on an OA that integrates software components from multiple producers or OSS components that are subject to different heterogeneous licenses, then we have the situation analogous to what we have presented in this paper, but now in the form of virtual SPLs from a virtual software production enterprise [26]. that spans multiple independent OSS projects and software production enterprises. As such, SPL concepts are compatible with OA systems that are composed from heterogeneously licensed components, but do not impact the formation or evolution of the software ecosystem niches where such systems may reside.

Our approach for using open software system architectures and component licenses as a lens that focuses attention to certain kinds of relationships within and across software supply networks, software ecosystems, and networks of software ecosystems has yet to be applied to systems on mobile computing platforms. Bosch [7] notes this is a neglected area of study, but one that may offer interesting opportunities for research and software product development. Thus, what happens when we consider Apple iPhone/iPad OS, Google Android OS phones, Nokia Symbian OS phones, Nokia Maemo OS hand-held computers, Microsoft Windows 7 OS phones, Intel Moblin OS netbooks, or Nintendo DS portable game consoles as possible platforms for OA system design and deployment? First, all of these devices are just personal computers with operating systems, albeit in small, easy to carry, and wireless form factors. They represent a mix of mostly proprietary operating system platforms, though some employ

Linux-based or other OSS alternative operating systems. Second, Mobile OS platforms owners (Apple, Nokia, Google, Microsoft) are all acting to control the software ecosystems for consumers of their devices through establishment of logically centralized (but possibly physically decentralized) application distribution repositories or online stores, where the mobile device must invoke a networked link to the repository to acquire (for fee/free) and install apps. Apple has had the greatest success in this strategy and dominates the global mobile application market and mobile computing software ecosystems. But overall, OA systems are not necessarily excluded from these markets or consumers. Third, given our design-time architecture of the case study system shown in Figure 5, is it possible to identify a build-time version that could produce a run-time version that could be deployed on most or all of these mobile devices? One such build-time architecture would compose an Opera Web browser, with Web services for word processing, calendaring and email, that could be hosted on either proprietary or OSS mobile operating systems. This alternative arises since Opera Software has produced run-time versions of its proprietary Web browser for these mobile operating systems, for accessing the Web via a wireless/cellular phone network connection. Similarly, in Figure 13 the instance architecture on the right could evolve to operate on a mobile platform like an Android-based mobile device or Symbian-based cell phone. So it appears that mobile computing devices do not pose any unusual challenges for our approach in terms of understanding their software ecosystems or the ecosystem niches for OA systems that could be hosted on such devices.

7. Conclusion

The role of software ecosystems in the development and evolution of heterogeneously-licensed open architecture systems has received insufficient consideration. Such systems are composed of components potentially under two or more licenses, open source software or proprietary or both, in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring. The software licenses of the components both facilitate and constrain

in which ecosystems a composed system may lie. In addition, the obligations and rights carried by the licenses are transmitted from the software component producers to system consumers through the architectural choices made by system integrators. Thus software component licenses help determine the contours of the software supply network and software ecosystem niche that emerge for a given implementation of a composed system architecture. Accordingly, we described examples for systems whose host software platform span the range of personal computer operating systems, Web services, and mobile computing devices.

Consequently, software component licenses and the architectural composition of a system determine the software ecosystem niche where a systems lies. Understanding and describing software ecosystem niches is a key contribution of this work. A case study of an open architecture software system that articulates different niches was employed to this end. We examined how the architecture and software component licenses of a composed system at design-time, build-time, and run-time helps determine the system's software ecosystem niche, and provides insight for identifying potential evolutionary paths of software system, architecture, and niches. Similarly, we detailed the ways in which a composed system can evolve over time, and how a software system's evolution can change or shift the software ecosystem niche in which the system resides and thus producer-consumers relationships. Then we described how virtual software product lines can be can be identified through a lens that examines the association between open architectures, software component licenses, and software ecosystems.

Finally, in related work [2, 3, 4] we identified structures for modeling software licenses and the license architecture of a system, and automated support for calculating its rights and obligations. Such capabilities are needed in order to manage and track an OA system's evolution in the context of its ecosystem niche. We have outlined an approach for achieving these structures and support and sketched how they further the goal of reusing and exchanging alternative software components and software architectural compositions. More work re-

mains to be done, but we believe this approach transforms a vexing problem of stating in detail how study of software ecosystems can be tied to core issues in software engineering like software architecture, product lines, component-based reuse, license management, and evolution, into a manageable one for which workable solutions can be obtained.

Acknowledgments

This research is supported by grants #0534771 and #0808783 from the U.S. National Science Foundation, and the Acquisition Research Program at the Naval Postgraduate School. No review, approval, or endorsement is implied.

References

[1] Alspaugh, T. A., Antón, A. I., 2008. Scenario support for effective requirements. Information and Software Technology 50 (3), 198–220.

[2] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., May 2009. Analyzing software licenses in open architecture software systems. In: 2nd International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS). pp. 1–4.

[3] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., 2009. Intellectual property rights requirements for heterogeneously-licensed systems. In: 17th IEEE International Requirements Engineering Conference (RE'09). pp. 24–33.

[4] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., 2009. The role of software licenses in open architecture ecosystems. In: First International Workshop on Software Ecosystems (IWSECO-2009). pp. 4–18.

[5] Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice. Addison-Wesley Longman.

[6] Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley.

[7] Bosch, J., 2009. From software product lines to software ecosystems. In: 13th International Software Product Line Conference (SPLC'09). pp. 111–119.

[8] Bosch, J., Bosch-Sijtsema, P., 2010. From integration to composition: On the impact of software product lines, global development and ecosystems. J. Syst. Softw. 83 (1), 67–76.

[9] Boucharas, V., Jansen, S., Brinkkemper, S., 2009. Formalizing software ecosystem modeling. In: First International Workshop on Open Component Ecosystems (IWOCE'09). pp. 41–50.

[10] Breaux, T. D., Anton, A. I., 2005. Analyzing goal semantics for rights, permissions, and obligations. In: RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering. pp. 177–188.

[11] Breaux, T. D., Anton, A. I., 2008. Analyzing regulatory rules for privacy and security requirements. IEEE Transactions on Software Engineering 34 (1), 5–20.

[12] Brown, A. W., Booch, G., 2002. Reusing open-source software and practices: The impact of open-source on commercial vendors. In: Software Reuse: Methods, Techniques, and Tools (ICSR-7). pp. 381–428.

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

[14] Dashofy, E. M., 2007. Supporting stakeholder-driven, multi-view software architecture modeling. Ph.D. thesis, University of California, Irvine.

[15] Feldt, K., 2007. Programming Firefox: Building Rich Internet Applications with XUL. O'Reilly Media, Inc.

[16] Firesmith, D., Jan.–Feb. 2004. Specifying reusable security requirements. Journal of Object Technology 3 (1), 61–75.

[17] German, D. M., Hassan, A. E., 2009. License integration patterns: Dealing with licenses mismatches in component-based development. In: 28th International Conference on Software Engineering (ICSE '09). pp. 188–198.

[18] Jansen, S., Brinkkemper, S., Finkelstein, A., 2009. Business network management as a survival strategy: A tale of two software ecosystems. In: First Workshop on Software Ecosystems. pp. 34–48.

[19] Jansen, S., Finkelstein, A., Brinkkemper, S., 2009. A sense of community: A research agenda for software ecosystems. In: 28th International Conference on Software Engineering (ICSE '09), Companion Volume. pp. 187–190.

[20] Jensen, C., Scacchi, W., Jul./Sep. 2005. Process modeling across the web information infrastructure. Software Process: Improvement and Practice 10 (3), 255–272.

[21] Kuehnel, A.-K., Jun. 2008. Microsoft, open source and the software ecosystem: of predators and prey—the leopard can change its spots. Information & Communucation Technology Law 17 (2), 107–124.

[22] Kuhl, F., Weatherly, R., Dahmann, J., 1999. Creating computer simulation systems: an introduction to the high level architecture. Prentice Hall.

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

[24] Meyers, B. C., Oberndorf, P., 2001. Managing Software Acquisition: Open Systems and COTS Products. Addison-Wesley Professional.

[25] Nelson, L., Churchill, E. F., 2006. Repurposing: Techniques for reuse and integration of interactive systems. In: International Conference on Information Reuse and Integration (IRI-08). p. 490.

[26] Noll, J., Scacchi, W., Feb. 1999. Supporting software development in virtual enterprises. J. of Digital Information 1 (4).

[27] Noll, J., Scacchi, W., 2001. Specifying process-oriented hypertext for organizational computing. J. Network and Computing Applications 24 (1), 39–61.

[28] Open Source Initiative, 2010. Open Source Definition. `http://www.opensource.org/docs/osd`.

[29] Oreizy, P., 2000. Open architecture software: A flexible approach to decentralized software evolution. Ph.D. thesis, University of California, Irvine.

[30] Ovaska, P., Rossi, M., Marttiin, P., 2003. Architecture as a coordination tool in multi-site software development. Software Process: Improvement and Practice 8 (4), 233–247.

[31] Rosen, L., 2005. Open Source Licensing: Software Freedom and Intellectual Property Law. Prentice Hall.

[32] Scacchi, W., 2007. Free/open source software development: Recent research results and emerging opportunities. In: 6th Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007). pp. 459–468.

[33] Scacchi, W., Alspaugh, T. A., May 2008. Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In: 5th Annual Acquisition Research Symposium.

[34] Taylor, R. N., Medvidovic, N., Dashofy, E. M., 2009. Software Architecture: Foundations, Theory, and Practice. Wiley.

[35] Unity Technologies, Dec. 2008. End User License Agreement. `http://unity3d.com/unity/unity-end-user-license-2.x.html`.

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

[37] Ven, K., Mannaert, H., 2008. Challenges and strategies in the use of open source software by independent software vendors. Information and Software Technology 50 (9-10), 991–1002.

[38] Yau, S. S., Chen, Z., 2006. A framework for specifying and managing security requirements in collaborative systems. In: Third International Conference on Autonomic and Trusted Computing (ATC 2006). pp. 500–510.