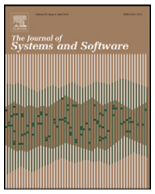
The Journal of Systems and Software 85 (2012) 1479–1494

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss



Understanding the role of licenses and evolution in open architecture software ecosystems

Walt Scacchi, Thomas A. Alspaugh*

Institute for Software Research, University of California, Irvine, USA

A R T I C L E   I N F O

Article history:
Received 30 April 2010
Received in revised form
21 December 2011
Accepted 19 March 2012
Available online 28 March 2012

Keywords:
Software architecture
Software ecosystems
Software licenses
Open source software
Software evolution

A B S T R A C T

The role of software ecosystems in the development and evolution of open architecture systems whose components are subject to different licenses 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 help to better define the *software ecosystem niche* in which a given system lies. Understanding and describing software ecosystem niches for open architecture systems is a key contribution of this work. An example 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.

© 2012 Elsevier Inc. All rights reserved.

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) (Oreizy, 2000), 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 (Fig. 1). With this approach, another organization often comes between software component producers and system consumers in order to compose and configure the produced components into a configured system. 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 realizes or instantiates 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 composed OA systems can achieve reuse benefits such as reduced costs, increased reliability, and potentially increased agility in evolving to meet changing needs. However, an emerging challenge is to realize the benefits of this approach when the individual components are *heterogeneously licensed* (Alspaugh et al., 2010; German and Hassan, 2009; Scacchi and Alspaugh, 2008), 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.

This challenge is inevitably entwined with the software ecosystems that arise for OA systems (Fig. 2). 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 composition and configuration of the OA of the system(s) in question,
- the open interfaces met by the components,

* Corresponding author.
E-mail addresses: wscacchi@ics.uci.edu (W. Scacchi), thomas.alspaugh@acm.org (T.A. Alspaugh).

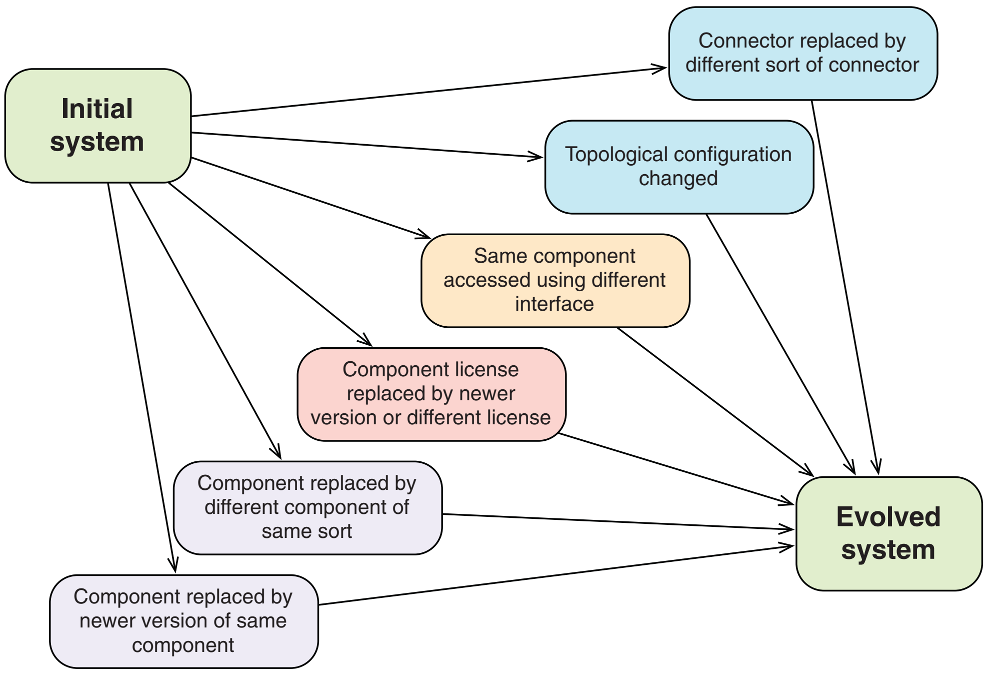

Initial system
Connector replaced by different sort of connector
Topological configuration changed
Same component accessed using different interface
Component license replaced by newer version or different license
Component replaced by different component of same sort
Component replaced by newer version of same component
Evolved system

Fig. 1. Paths of evolution for an OA system (Section 6).

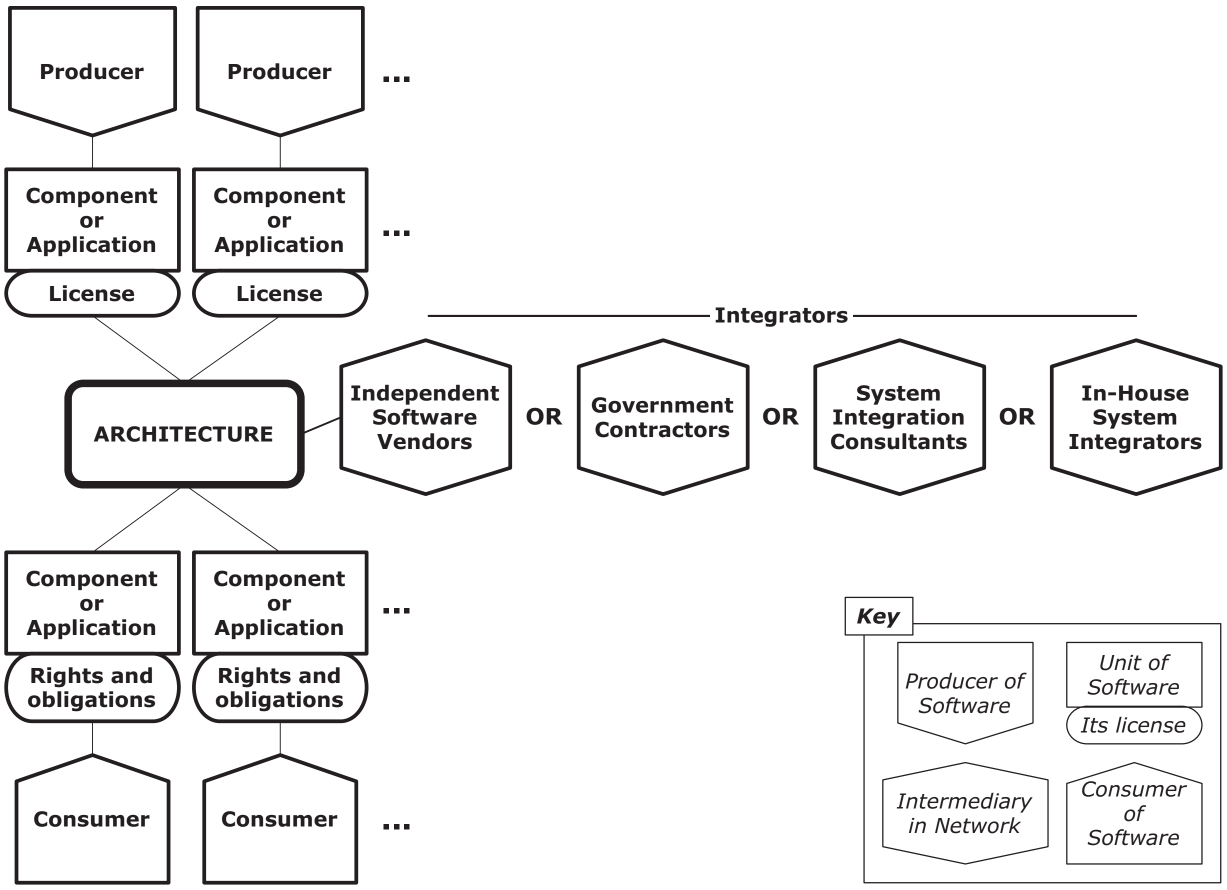


Fig. 2. Schema for OA software supply networks (notation follows Boucharas et al., 2009).

- 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.

These four items play a key role in defining the *software ecosystem niche* for a specific configured system—the specific software supply network that interconnects particular software producers of specific components, integrators, the software system architecture and its configured instantiation, and its consumers—as the remainder of this paper will make clear.

In our previous work (Alspaugh et al., 2009a,b, 2011, 2010; Scacchi and Alspaugh, 2008) we examined how software licenses interact in significant ways through the software architecture of the system. Our approach, implemented in an Eclipse-based software architecture environment, automatically evaluates license conflicts in a software architecture and calculates the *virtual license* of rights and obligations for a composed system that result when its constituent components are licensed heterogeneously. With it architects directly examine the design decisions' licensing consequences: in the decision context, with enough information to identify definite license conflicts rather than only potential license conflicts, and early enough in the development process to make the right decision rather than correct a wrong one. This work contrasts with much practice and other research in which a configured system is examined after the fact, and often with substantial manual work by experts, to determine what licensing conflicts might exist in it. Here we build on our previous work by extending its context from software architecture to software ecosystems. The ecosystem context allows architects and integrators to examine potential evolution paths and the consequences of each one, with the ability to steer that evolution by specific changes to the system architecture and build- and run-time configuration.

The remainder of this paper is organized as follows. Section 2 motivates the work through a sequence of examples. Section 3 places this work in the context of related research. Section 4 discusses open architecture, and Section 5 discusses the ecosystems that arise around open architecture systems. Section 6 addresses evolution of software ecosystems. Section 7 discusses some implications that follow from this study, and Section 8 concludes the paper. Background on kinds of software licenses is presented in Appendix.

2. Motivating examples

2.1. Firefox: monolithically licensed

A few years ago, it was typical for a software system to be subject to a single intellectual property (IP) or copyright license covering the entire system, especially for proprietary software systems, but even for open source software (OSS). An example is the globally popular Firefox web browser, whose OSS is subject to the Mozilla Public License (MPL) version 1.1 (OSI, 2011). More recently, the Mozilla organization has updated its licensing strategy so that new OSS it produces is "tri-licensed." This allows a licensee the choice to access, modify, and redistribute these systems under terms and conditions specified in either MPL, the GNU Project's General Public License (GPL), or the Lesser General Public License (LGPL) (OSI, 2011), while Firefox as a software product is under MPL. Users of Firefox and developers utilizing Firefox as a single component of larger systems need not concern themselves with whether the Mozilla organization has sufficient legal rights to all the Firefox code; Mozilla has assumed that responsibility.

2.2. Unity: heterogeneously licensed, closed architecture

These days, a growing segment of software systems are subject to multiple licenses, some of which may indicate potentially conflicting terms and conditions in different licenses, rather than to a single monolithic license. For example, the Unity game development tool, produced by Unity Technologies, is subject to multiple licenses (Unity Technologies, 2008). Its license agreement, from which we quote below, comprises a proprietary license for the core Unity software and presumably for the entire Unity system, plus at least 15 distinct licenses for at least 26 externally produced components, groups of components, and libraries, at least one of which has been further extended by Unity:

1. The Mono Class Library, Novell, Inc., MIT license,
2. The Mono Runtime Libraries, Novell, Inc., LGPLv2 (updated),
3. Boo, Rodrigo B. Oliveira, BSD license variant,
4. UnityScript, Rodrigo B. Oliveira, BSD license variant,
5. PhysX physics library, Novell Inc., proprietary,
6. libvorbis, Xiph.org Foundation, BSD license variant,
7. libtheora, Xiph.org Foundation, BSD license,
8. zlib general purpose compression library, Jean-loup Gailly and Mark Adler, inferred zlib/libpng license,
9. libpng PNG reference library, three individuals and Group 42 Inc., inferred zlib/libpng license,
10. jpeglib JPEG library, Thomas G. Lane, custom OSS license,
11. Twilight Prophecy SDK, Twilight 3D Finland Oy Ltd., inferred zlib/libpng license,
12. dynamic_bitset, Chuck Allison and Jeremy Siek, custom OSS license,
13. The Mono C# Compiler and Tools, Novell, Inc., GPLv2 updated,
14. libcurl, Daniel Stenberg, MIT license derivative,
15. PostgreSQL Database Management System, University of California and PostgreSQL Development Group, BSD license derivative,
16. FreeType, The FreeType Project, FreeType License,
17. NVIDIA Cg compiler, NVIDIA Corp., GPLv2,
18. Scintilla and SciTE (source code editing), Neil Hodgson, Scintilla License,
19. 7-Zip Command (source code editing), Igor Pavlov, LGPLv2 (updated),
20. AES code (encryption/authentication), Brian Gladman, BSD license derivative,
21. FreeImage library, FreeImage project, FreeImage Public License,
22. Little CMS color management engine, Marti Maria Saguer, MIT license,
23. paintlib, Ulrich von Zadow and others, paintlib license,
24. Ericsson Texture Compression, Ericsson, proprietary license,
25. Particle Trimmer, Emil Persson, custom OSS license,
26. MonoDevelop IDE, MonoDevelop project and Unity, MIT license.

The overall software product license grants the right to install and use Unity but no rights to view or modify its source code (except for those components that are open source) or its design artifacts. Ordinarily the use of a properly licensed copy is unrestricted unless the software is patented; it is not clear whether any of Unity is patented or not, but as is often the case for proprietary licenses the Unity license states that unlicensed use is prohibited. Parts of the license explicitly give the user responsibility for obtaining any licenses required for (presumably future) patents that the software may infringe; trademarks are not mentioned except when reserving rights to them. Furthermore, an external developer or integrator has no access to Unity's architecture, and so cannot tell whether/how the separate license obligations for the externally produced components propagate to the obligations for Unity as a whole. However, the presence of a component with a reciprocal

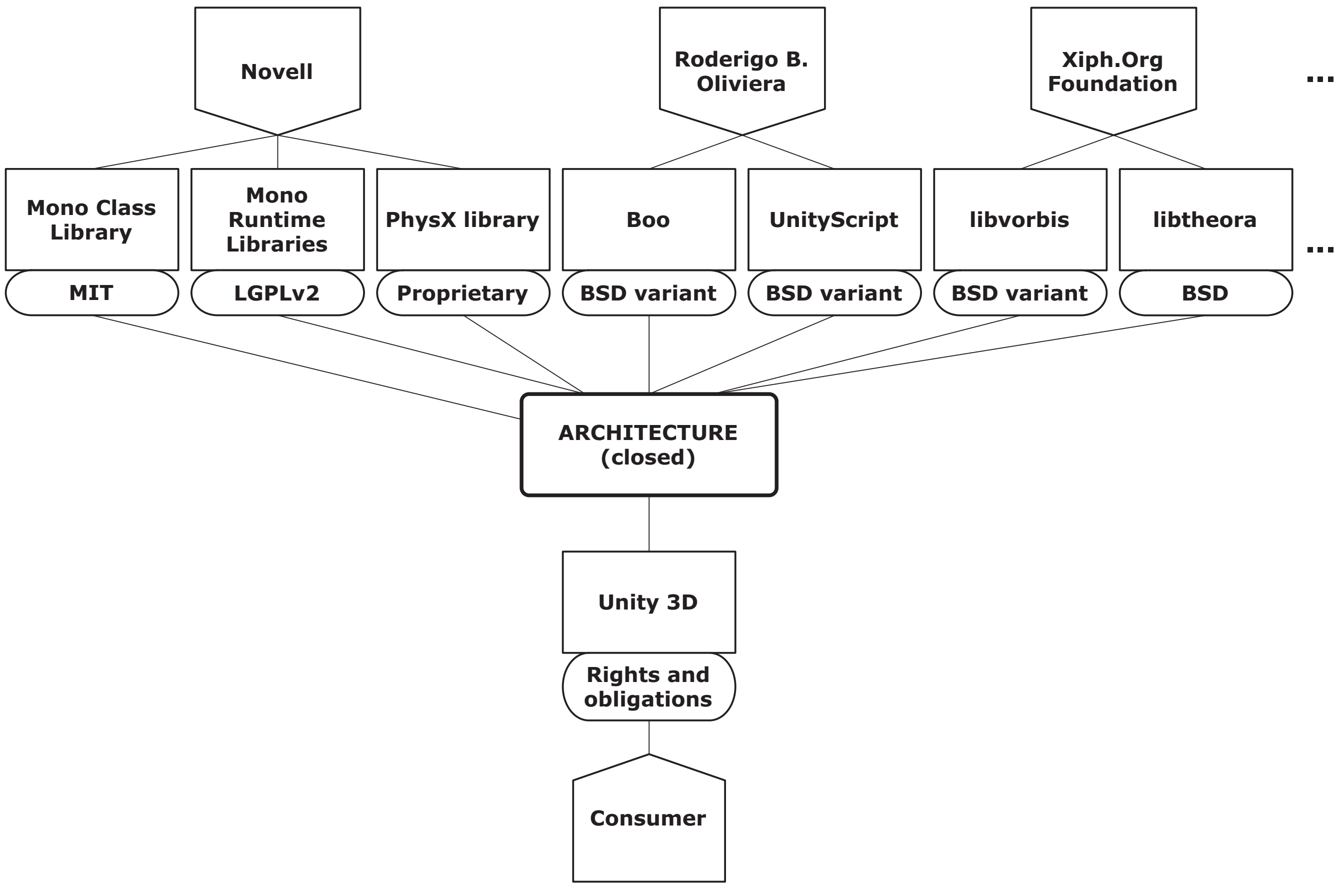


Fig. 3. Ecosystem for Unity game development tool (partial).

license that can propagate obligations to other components (17: NVIDIA Cg compiler, GPLv2) raises the necessity for Unity Technologies to have addressed these obligations architecturally in order for an end user not to propagate them further if using Unity as a component of a larger system.

The software ecosystem for Unity as a standalone software package is delimited by the diverse set of software components listed above (Fig. 3). However the architecture that integrates and configures these components is closed: the architecture has not been made public, and much of the system is proprietary so that even what could be inferred from the source code cannot be determined. Thus consumers cannot determine 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. Since there are several interpretations of some important OSS license provisions, this may be significant; did Unity Technologies firewall GPLv2's propagating obligations with dynamic links (following one well-supported interpretation of GPLv2) or more strongly with client-server connections (following another well-supported but more cautious interpretation)? A development organization with their own legal interpretation of GPLv2 and considering using Unity as a critical element of a composed system may need to know.

As a consequence, there are several important questions that can't be answered about this ecosystem, but that an open architecture ecosystem annotated with software licenses and connector types can and should answer.

- What is Unity's *virtual license*, the set of rights available for the entire system and obligations demanded in exchange for those rights (Alspaugh et al., 2009b)?
- What portions of Unity do the various listed licenses pertain to, especially licenses such as the GNU General Public License that can propagate obligations along architectural connections to other components?
- What components of Unity can be evolved to later versions or replaced by similar components, in order to evolve the system toward more desirable functionality, desired software qualities, or more advantageous ecosystem and system evolution possibilities?
- For each component, how much of that component is being used by Unity? In other words, what interface is Unity using the component through? What other components support that interface, and what shims are available or could be developed to bridge the gap between that interface and the interfaces of other desirable components and versions?
- How and to what extent is Unity vulnerable to:
  - potential litigation for license violations for copyright or copyleft infringements, or
  - coercion due to dependence on specific development libraries and development or configuration tools?

There are also questions that cannot be answered even for an OA ecosystem, due to the differences between copyright law, under which an author gains specific exclusive rights for a specific term of years by the act of creation, and patent law, under which other inventors may unpredictably be granted new exclusive rights in the future over previously unencumbered parts of others' software systems, and also with trademark law, whose provisions are temporally dynamic and less uniform internationally.

- How and to what extent is Unity vulnerable to threats of patent infringement suits, whether for actual infringement, to force a settlement to avoid a lengthy, expensive, and risky court battle, or to persuade a system/platform vendor to engage in a cross-licensing agreement along with payment of license fees?
- How and to what extent is Unity vulnerable to co-opting of needed trademarks in some jurisdiction?

2.3. Google Chrome: heterogeneously licensed, open interfaces

The Google Chrome web browser represents yet another software ecosystem whose boundaries are defined in part through its use of externally licensed OSS components, that can be compared to Firefox and Unity. The license for Google's Chromium project (Chromium, 2011), from whose code base the Google Chrome browser is primarily built, comprises the BSD license for the Chromium core developed specifically for Google Chrome, plus 27 external components and libraries (some used only for specific platforms) under 14 distinct licenses:

1. bsdiff, BSD Protection License,
2. bspatch, BSD Protection License,
3. bzip2, BSD License,
4. dtoa, BSD License
5. ffmpeg, LGPL
6. HarfBuzz, MIT License,
7. hunspell, MPL 1.1 or GPL 2.0 or LGPL,
8. ICU, ICU License
9. JSCRE, BSD License,
10. libevent, BSD License,
11. libjpeg, libjpeg License,
12. libpng, libpng License,
13. libxml, MIT License,
14. libxslt, MIT License,
15. LZMA SDK, Special Exception License,
16. modp_b64, BSD License,
17. Mozilla interface to Java Plugin APIs, MPL 1.1 or GPL 2.0 or LGPL,
18. npapi, MPL 1.1 or GPL 2.0 or LGPL,
19. nspr, MPL 1.1 or GPL 2.0 or LGPL,
20. nss, MPL 1.1 or GPL 2.0 or LGPL,
21. Pthreads for win32, LGPL 2.1,
22. Skia, Apache License 2.0,
23. sqlite, Public domain dedication,
24. V8 assembler, BSD License,
25. WebKit, BSD or LGPL 2 or LGPL 2.1,
26. WTL, Microsoft Public License (Ms-PL),
27. zlib, zlib License.

Two of the libraries (libpng and zlib) are also used by Unity though possibly under different licenses, and one component (LZMA) is part of a Unity component (7-Zip).

An examination of the component licenses shows that no Chromium component is subject to a proprietary license (Ms-PL, despite its name, is a permissive open source license) and every one of the external Chromium components is available under a license that does not propagate license obligations to other components. Every component that is licensed under GPL, which can propagate obligations to other components depending on the connectors and architectural configuration around them, is also available under a non-propagating license such as MPL. It is evident that Google has chosen a policy of avoiding components licensed only under GPL and similar reciprocal licenses, forgoing the much broader selection of GPL-licensed components (approximately half of all open-source software is licensed under GPLv2) in exchange for not needing to consider architectural interactions among components, or whether any subsequent development or integration of Chromium can virally propagate GPLv2 obligations into other systems or applications.

It appears that all the external components have open interfaces (i.e. public and standardized), so that Chromium can evolve by replacing components with others implementing the same interfaces, or shimmed to them, as long as the replacements are also under non-propagating OSS licenses. However, Chromium's overall architectural composition, its architectural design, is (to our knowledge) not open and perhaps not even explicit.

2.4. Discussion

Firefox, Unity, and Google Chrome have illustrated three related approaches to software licenses, software architecture, and software ecosystems.

- Firefox is a monolithically licensed OSS system: all its code is given to the project under contributor license agreements (Jensen and Scacchi, 2011) that support releasing the entire project under a single license. External components are kept at arm's length, architecturally speaking, as plug-ins subject to their own licenses, with no license interaction with Firefox itself.
- Unity is a closed system with externally produced components, some with open interfaces and OSS licenses and others with proprietary licenses. The external components retain their own licenses which are incorporated into the overall license for Unity either by reference or by quoting the license text. Unity Technologies has likely followed an internal, manual process for resolving potential license conflicts among components, so that it can offer Unity to its consumers without causing the suppliers of those components to object. Because Unity Technologies does not release Unity overall as an OSS project, most of the sublicensing provisions of the components' licenses do not come into play, simplifying Unity Technologies' manual analysis for license conflicts at the expense of preventing licensees from modifying Unity to meet their own needs more exactly. Some of the components are OSS, and for one of them (26: MonoDevelop IDE) Unity Technologies' modifications are open as well, but Unity users cannot modify components themselves and rebuild a more capable version of Unity from them.
- Google Chrome is an OSS system incorporating externally produced components. However, it is not an OA system since it does not appear to have a formally specified open architecture that explicitly composes components interlinked through connectors to derive or realize a buildable system configuration. Instead, as in most OSS projects, its parent project Chromium relies on an implicit architecture that cannot be completely identified and may only be assessed by reading the source code and reviewing online artifacts and developer interaction records (e.g. postings to a bulletin board, reviewing bug reports, checking comments in source compilation build scripts, or developer chat channels).

  Because its architecture is implicit, the overall system license for Chromium cannot be calculated automatically (Alspaugh et al., 2009b, 2011) but is instead compiled manually. The several-years-old date of 2008 for the Chromium license, and the project's discussion of the change from JSCRE to the V8 regular expression engine (Chromium issues, 2009), for which the license was not updated, support this inference.

  In order to simplify the process for resolving potential license conflicts, the Chromium project appears to have limited its external components to those available under non-propagating, non-reciprocal OSS licenses. The Chromium source is publicly available for perusal and modification, but not under a single monolithic license; each component is licensed under its own. Users can modify and rebuild Chromium to suit their own needs, as long as they meet the (separate) license obligations of all the components, and do not contravene Google trademark or branding restrictions.

In summary, none of these widely used systems provide enough information to completely evaluate potential evolution paths, or to automatically calculate overall IP rights and obligations. But this information and IP stipulations are needed to fully articulate the software supply networks that reveal which software ecosystem instances (or niches) each system exists within. In order to explore the issues raised by open architecture software ecosystems, it is necessary to consider a system about which the necessary information is available. We do not claim that only open architecture systems are important or useful, but rather that only such systems can take full advantage of the evolutionary and analytical opportunities OAs support. Because it is not possible, in general, to infer a system's software architecture after the fact, or to satisfactorily impose an OA on a system developed without one, the system must be designed from the beginning with an explicit architecture as a first-class development architecture. Consequently, we present an example system that utilizes a simple, archectypal open architecture in Section 5. This system illustrates the issues that arise with more complex systems like Unity and Google Chrome, as well as additional possibilities not available without an OA, and does so with greater brevity and clarity.

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.

3. Related research

3.1. Software ecosystems

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 (Messerschmitt and Szyperski, 2003). Bosch (2009) builds a conceptual lineage from software product line (SPL) concepts and practices (Bosch, 2000; Clements and Northrop, 2001) 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 (Noll and Scacchi, 1999). 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).

3.2. Relations among and within software ecosystems

Jansen et al. (2009a,b) observe 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 (2007) for example, identifies that the world of OSS development is a loosely coupled collection of software ecosystems 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 (Jensen and Scacchi, 2005).

Boucharas et al. (2009) 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 (2008) examines how Microsoft's software ecosystem developed around 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.

OSS ecosystems also exhibit strong relationships between the ongoing evolution of OSS systems and their developer and user communities, such that the success of one co-depends on the success of the other (Scacchi, 2007). 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 (Ven and Mannaert, 2008).

3.3. Software ecosystems and software product lines

Along with other colleagues (Bosch and Bosch-Sijtsema, 2010; Brown and Booch, 2002; van Gurp et al., 2010), Bosch also 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 (Noll and Scacchi, 1999, 2001), as a key facet that can enable software producers to build systems from diverse sources.

3.4. Building on related work

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 (Bosch, 2000; Taylor et al., 2009). 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 (Ovaska et al., 2003). 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 (Scacchi and Alspaugh, 2008). We then find value in attributing open architecture components with their IP licenses (Alspaugh et al., 2009b), 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.

4. Open architectures

Open architecture (OA) is a software design customization technique introduced by Oreizy (2000) 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. Unity Technologies, 2008). Using this approach can lower development costs and increase reliability and function (Scacchi and Alspaugh, 2008). 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 explicitly interconnected components that are either open source or proprietary with open APIs, 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 (Alspaugh and Antón, 2008; Scacchi and Alspaugh, 2008). Thus, as noted earlier, neither Firefox, Unity, nor Google Chrome are OA systems, even though all three are built with OSS components. But how can we specify and design a system so that it does have an OA?

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed (Bass et al., 2003).

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 (Feldt, 2007) or "mashups" (Nelson and Churchill, 2006). 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 has no explicit license 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 (Rosen, 2005). If proprietary, they often cannot be redistributed, and so such components will be present in the design-, build-, 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" (Meyers and Oberndorf, 2001). Open APIs are not and cannot be licensed, but they can limit the propagation 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 (Kuhl et al., 1999), 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 license obligations.

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 (Alspaugh et al., 2009b; German and Hassan, 2009); 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' choice to select components that either ensure or avoid certain licenses (for example Firefox's policy of only accepting source code that can be tri-licensed, Section 2.1, or Google Chromium's apparent policy of excluding components governed by proprietary or strong-copyleft licenses, Section 2.3), 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 integrator 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.

5. Understanding open architecture software ecosystems

A software ecosystem constitutes a software supply network that connects software producers to integrators to consumers, through licensed components and composed systems. Fig. 4 illustrates a software ecosystem for an OA example system discused below. By analogy to Hutchinson's definition of a niche in a biological ecosystems as "an *n*-dimensional hypervolume . . . every point in which corresponds to a state of the environment which would permit the species . . . to exist indefinitely" (Hutchinson, 1957), we define software ecosystem niches below.

5.1. Software ecosystem niches

A *software ecosystem niche* articulates a specific software supply network that interconnects particular software producers of specific components, integrators, and consumers. The niche defined by a software system may lie within an existing single ecosystem, or it may span a network of several software producer ecosystems.

Firstly, a composed software system architecture largely determines the system's 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 and usage rights passed from the participating software component producers, through integrators, on to system consumers. Thus, system architects or component integrators help determine in which software ecosystem niche a given instance architecture for the system participates.

Secondly, system integrators can update or modify system architectural choices not only at design time, but also at build time, when components are joined together into an executable, or at run

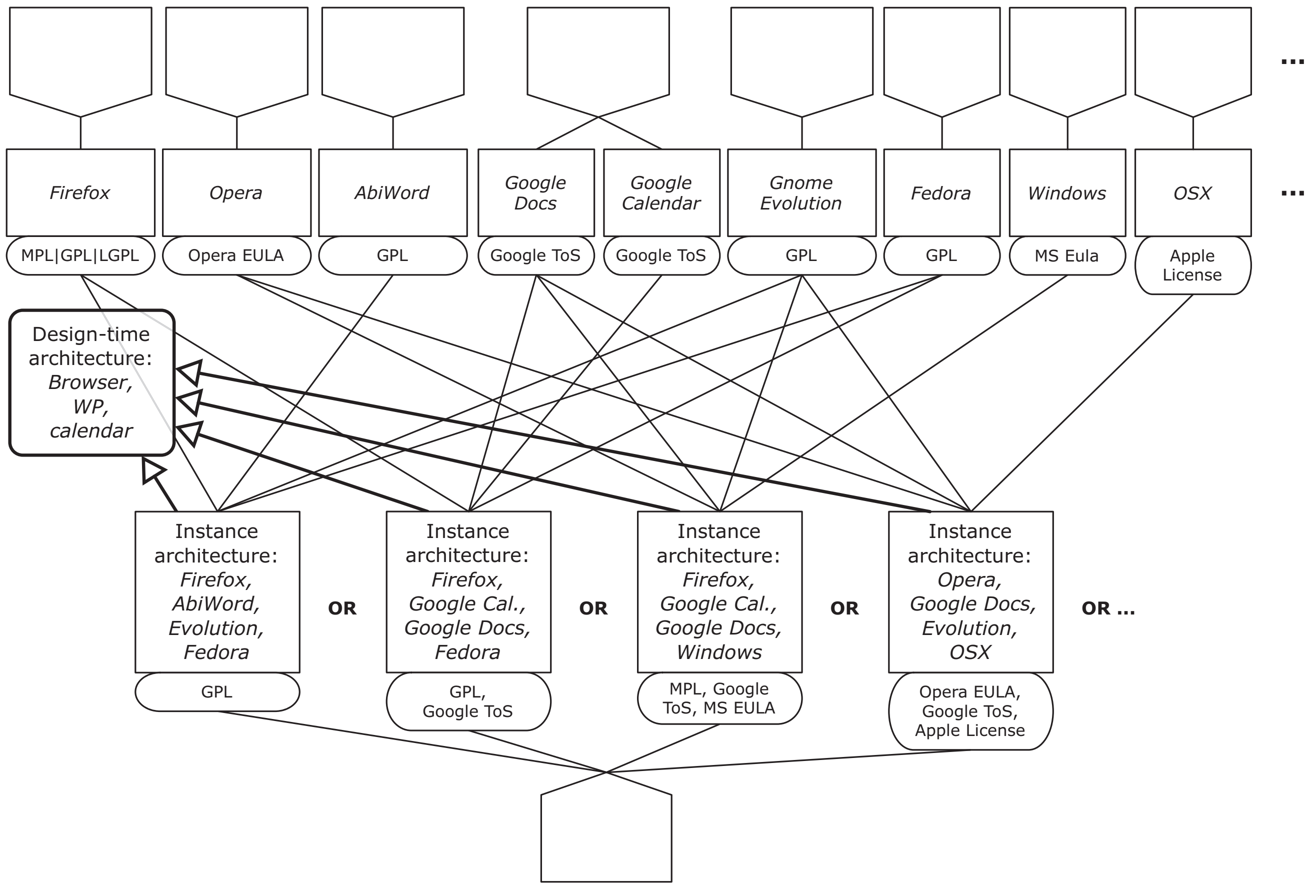

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

Fig. 4. Ecosystem for three possible instantiations of a single design architecture.

time, when bindings to remote executable services are instantiated, thus shifting the instantiated system to a related but distinct niche.

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 and how it can be advantageously steered for the future. Such a change implies at minimum that the software supply network may have been reconfigured, and thus obligations and rights passed from producers and integrators to system consumers may have also changed in some way. A system may evolve because its consumers want to migrate to alternatives from different component producers, or choose components whose licenses are now more 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. 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. It also helps articulate how the obligations and rights provided by producers are propagated/constrained by integrators onto system consumers as the system is developed and evolves.

5.2. An example system

To help explain how OA systems articulate software ecosystem niches, we provide a software architecture example system for use in this paper. This OA system utilizes a simple architectural design that composes a web browser, word processor, calendaring, and email applications, onto a host platform operating system, possibly with remote services for some components, designed and integrated by some organization and distributed to its consumers, some of whome may in turn integrate it into a larger system The same issues arise as if it utilized a graphics library, encryption module, typesetting engine, and thread management component instead, or with 400 components rather than 4, but this architecture illustrates the issues more simply and has the advantage of applying to many existing systems, including systems built by the authors. 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 Fig. 4. This design-time architecture appears in Fig. 5; note that it only specifies components by type rather than

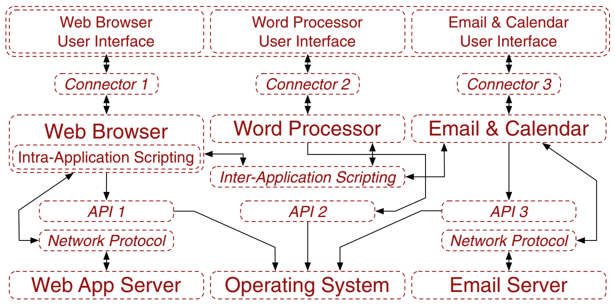


Fig. 5. A design-time architecture.

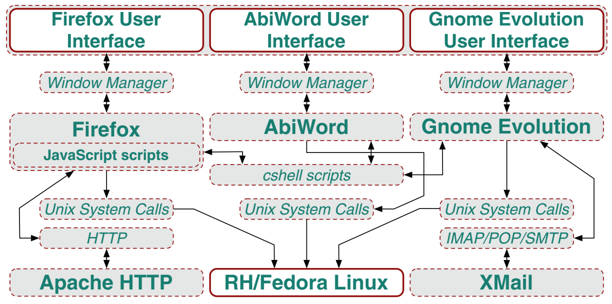


Fig. 6. A build-time architecture.

by producer, meaning the choice of producer component remains unbound at this point.

Then in Fig. 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 Fig. 7.

Figs. 7–9 display alternative run-time instantiations of the design-time architecture of Fig. 5. The architectural run-time instance in Fig. 7 corresponds to the software ecosystem niche shown in Fig. 10; Fig. 8 corresponds to the niche in Fig. 11; and Fig. 9 designates yet another niche different from the previous two. The run-time instantiations are then distributed to the consumers of the system.

This system's ecosystem is complex in important ways:

- *Alternatives exist for each component* that bring into play diverse possibilities for licenses, evolution paths, system capabilities, requirements, and ecosystems, such as MS Word (proprietary), AbiWord (OSS), or Google Docs (remote service) for the word processor.
- Some component choices *co-evolve with coordination among suppliers* (such as Mozilla and Gnome components) while others do not (Section 6).
- The system in its current open architecture is *independent of any one supplier*. Such ecosystems are more revealing and offer more evolution paths for study (and use) than a system in an ecosystem dominated by a single vendor such as Microsoft, Oracle, or SAP. Single-vendor-dominated ecosystems may be larger, but are less diverse and thus less interesting and offer fewer choices with significant ecosystem impact.
- The system is *independent of any one platform*; for example, it could be evolved by component replacement to run on a mobile device as discussed in Section 7, moving it into a much different niche.

The system can be instantiated with components all governed by the same license (as in Fig. 10), resulting in a monolithically licensed system like Firefox; and it can be instantiated with diversely licensed components (as in Fig. 11), resulting in a heterogeneously licensed system like Unity and Google Chrome. Unlike those three, however, it also is an OA system and so its virtual license can be calculated and its software ecosystem niche can be directly studied and evolved toward a more desirable one. Because it is OA, it offeres more choices of components and configurations, and thus more possible niches, along with more ways to move among and take advantage of them; all that Firefox, Unity, and Google Chrome offer as expository examples, plus more.

The insights provided by the example system allow one, we believe, to anticipate or even predict the kinds of issues that will arise when new platforms emerge.

The four primary components collectively represent more than a million lines of code. Each component, and its subcomponents recursively down to the smallest, is a composition of other more primitive components that may be independently developed or developed as part of this system, and may be added to the ecosystem relationships in order to consider its effect on supply chains and evolution. An individual component such as Firefox constitutes a micro-platform itself on which Ajax, Rich Internet Applications, or other scripted functionality (e.g. invoking an embedded link to a YouTube Video player) can run internally, constituting an embedded ecosystem. 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).

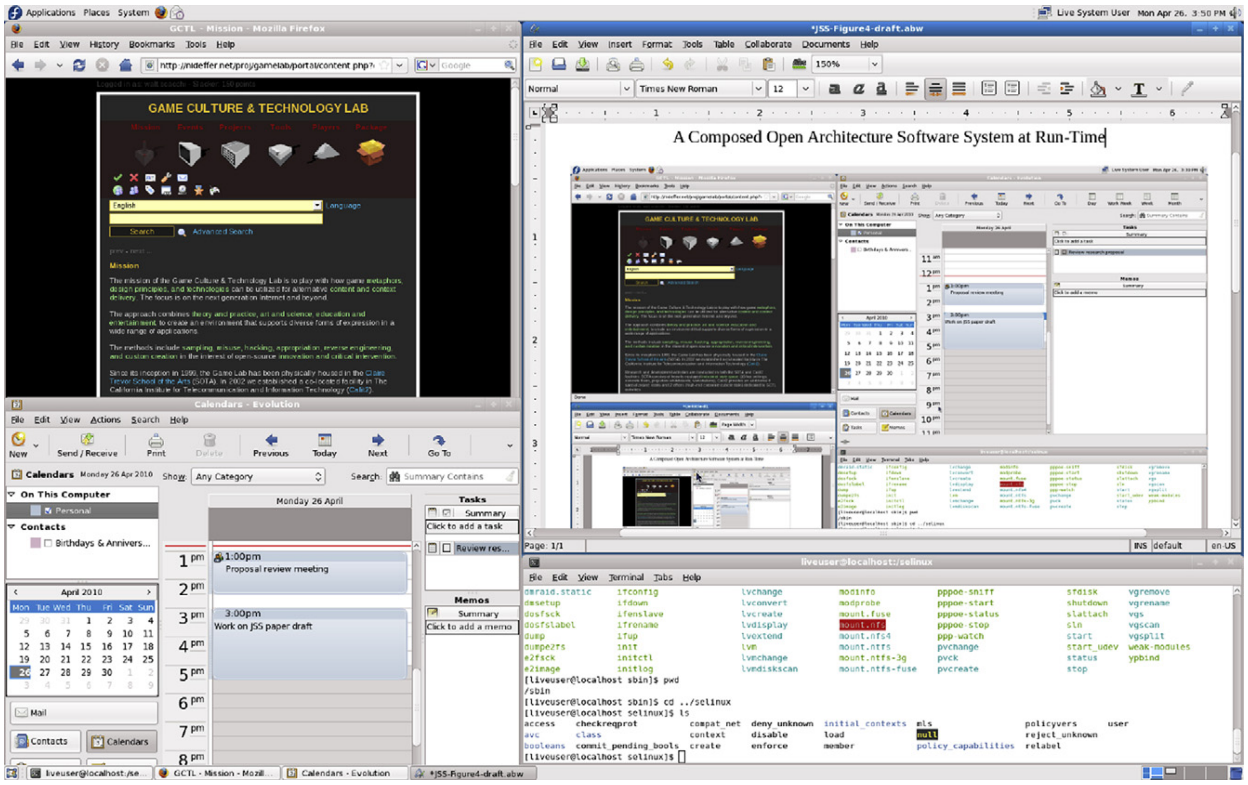
The OA we describe covers a number of systems we have identified, built, and deployed in a university research laboratory, and as far as can be externally determined also many distinct systems integrated by organizations and distributed internally or to a customer base. We have also developed OA systems with more complex architectures that incorporate 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. Even these more complex OAs still reflect the core architectural concepts and constructs, software ecosystem relationships, challenges, and solutions that we present more accessibly in our example system.

The software ecosystem niches for the example 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 (Alspaugh et al., 2010; German and Hassan, 2009; Scacchi and Alspaugh, 2008) by examining how OA development 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 niche of an open architecture software system:

- It must rest on a license structure of rights and obligations, focusing on obligations that are enactable and testable (Alspaugh et al., 2009b, 2010).[1]
- It must take account of the distinctions between the design-time, build-time, and distribution-time architectures (Section 4) 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?



A Composed Open Architecture Software System at Run-Time

Mission

The mission of the Game Culture & Technology Lab is to play with how game metaphors, design principles, and technologies can be utilized for alternative content and context delivery. The focus is on the next generation Internet and beyond.

The approach combines theory and practice, art and science, education and entertainment, to create an environment that supports diverse forms of expression in a wide range of applications.

The methods include sampling, misuse, hacking, appropriation, reverse engineering, and custom creation in the interest of open-source innovation and critical intervention.

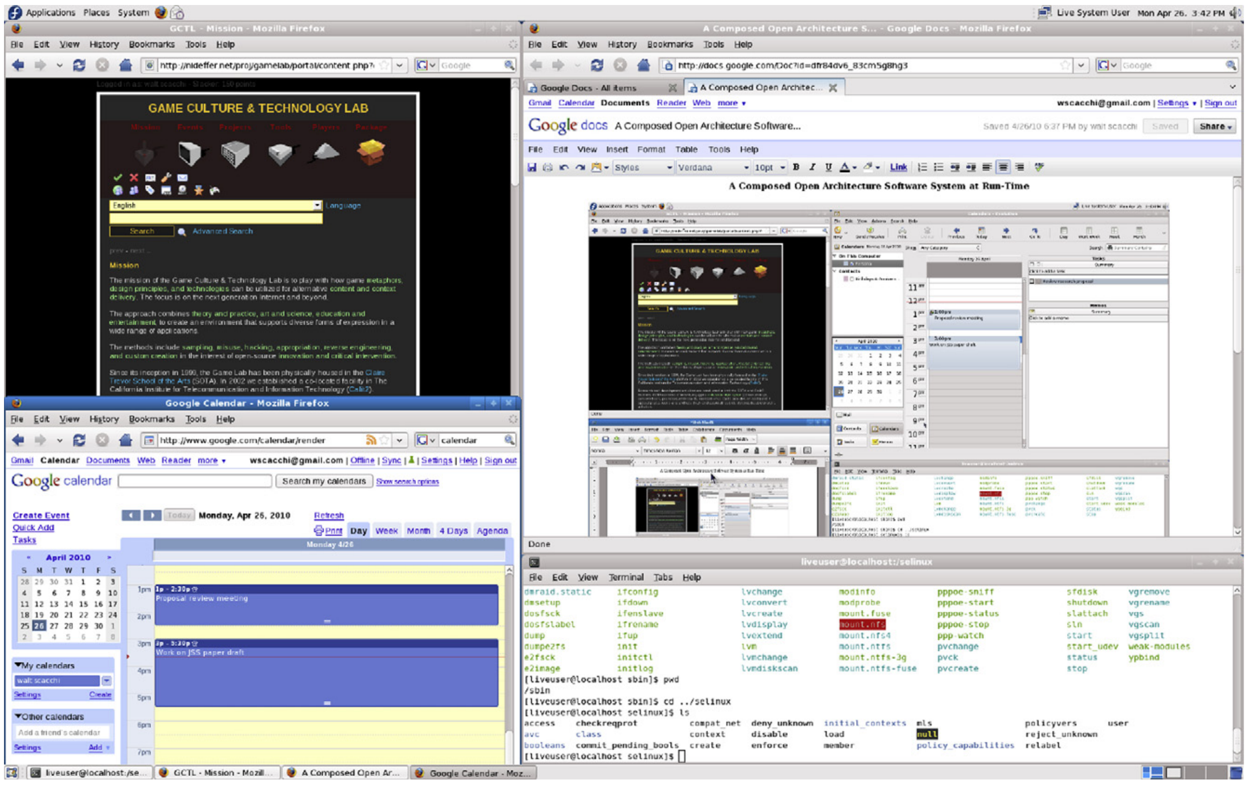
Since its inception in 1999, the Game Lab has been physically housed in the Claire Trevor School of the Arts (SOTA). In 2002 we established a co-located facility in The California Institute for Telecommunication and Information Technology (Calit2).

1:00pm Proposal review meeting

3:00pm Work on JSS paper draft

```
dmraid.static  ifconfig    lvchange    modinfo          pppoe-sniff   sfdisk      vgremove
dmsetup        ifdown      lvconvert   modprobe         pppoe-start   shutdown    vgrename
dosfsck        ifenslave   lvcreate    mount.fuse       pppoe-status  slattach    vgs
dosfslabel     ifrename    lvdisplay   mount.nfs        pppoe-stop    sln         vgscan
dump           ifup        lvextend    mount.nfs4       ppp-watch     start       vgsplit
dumpe2fs       init        lvm         mount.ntfs       pvchange      start_udev  weak-modules
e2fsck         initctl     lvmchange   mount.ntfs-3g    pvck          status      ypbind
e2image        initlog     lvmdiskscan mount.ntfs-fuse  pvcreate      stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot          compat_net  deny_unknown  initial_contexts  mls                   policyvers      user
avc       class                 context     disable       load              null                  reject_unknown
booleans  commit_pending_bools  create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$ 
```

Fig. 7. An instantiation at run time (Firefox, AbiWord, Gnome Evolution, Fedora) of the build-time architecture of Fig. 6 that determines the ecosystem niche of Fig. 10.


GCTL - Mission - Mozilla Firefox
A Composed Open Architecture S... - Google Docs - Mozilla Firefox
http://docs.google.com/Doc?id=dfr84dv6_83cm5g8hg3
Google Docs - All items
A Composed Open Architec...
Google docs
Saved 4/26/10 5:37 PM by walt scacchi
Share
Google Calendar - Mozilla Firefox
http://www.google.com/calendar/render
wscacchi@gmail.com
Google calendar
Search my calendars
Create Event
Quick Add
Tasks
Today
Monday, Apr 26, 2010
Refresh
Print
Day Week Month 4 Days Agenda
April 2010
My calendars
walt scacchi
Other calendars
Add a friend's calendar
Done
liveuser@localhost:/selinux

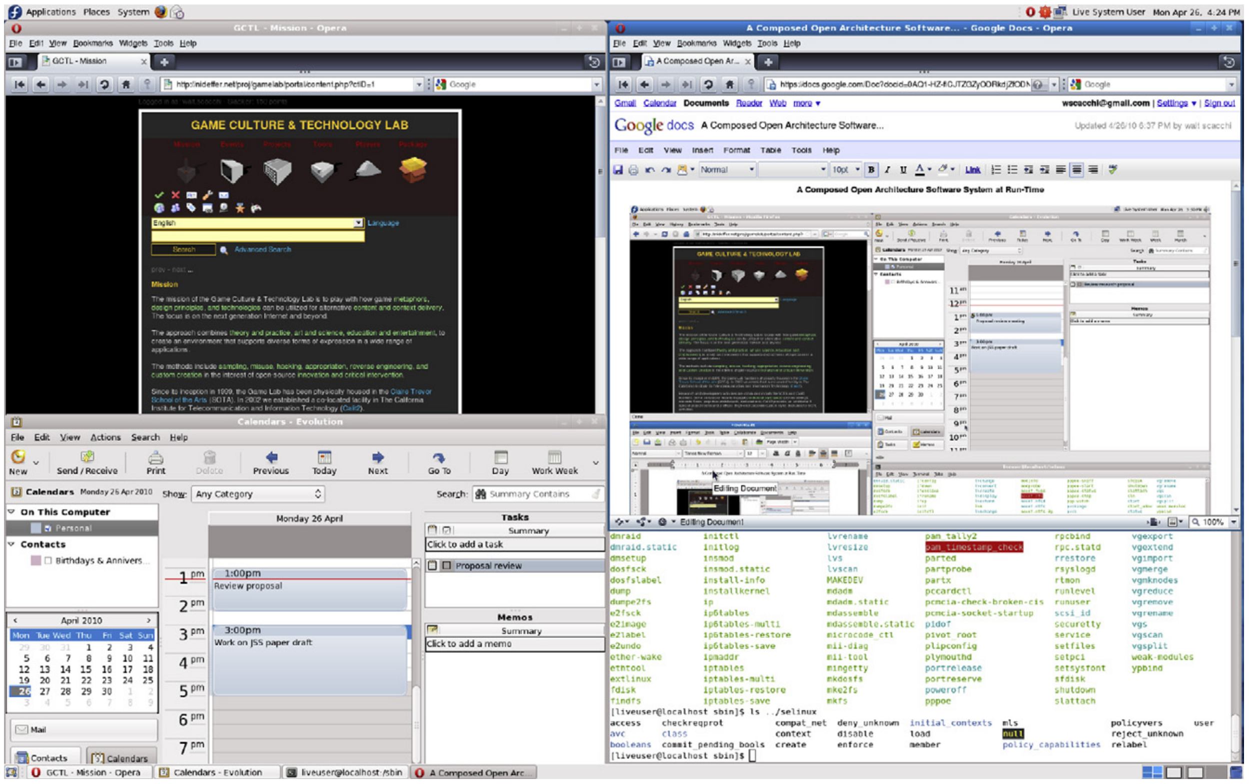
A Composed Open Architecture Software System at Run-Time

1p - 2:30p Proposal review meeting

3p - 5:30p Work on JSS paper draft

```
dmraid.static  ifconfig    lvchange    modinfo          pppoe-sniff   sfdisk      vgremove
dmsetup        ifdown      lvconvert   modprobe         pppoe-start   shutdown    vgrename
dosfsck        ifenslave   lvcreate    mount.fuse       pppoe-status  slattach    vgs
dosfslabel     ifrename    lvdisplay   mount.nfs        pppoe-stop    sln         vgscan
dump           ifup        lvextend    mount.nfs4       ppp-watch     start       vgsplit
dumpe2fs       init        lvm         mount.ntfs       pvchange      start_udev  weak-modules
e2fsck         initctl     lvmchange   mount.ntfs-3g    pvck          status      ypbind
e2image        initlog     lvmdiskscan mount.ntfs-fuse  pvcreate      stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot          compat_net  deny_unknown  initial_contexts  mls                   policyvers      user
avc       class                 context     disable       load              null                  reject_unknown
booleans  commit_pending_bools  create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$ 
```

Fig. 8. A second instantiation at run time (Firefox, Google Docs and Calendar, Fedora) determining the ecosystem niche of Fig. 11.


GCTL - Mission - Opera
http://nideffer.net/proj/gamelab/portal/content.php?ctID=1
A Composed Open Architecture Software... - Google Docs - Opera
Calendars - Evolution
Editing Document
100%

GAME CULTURE & TECHNOLOGY LAB

Mission

The mission of the Game Culture & Technology Lab is to play with how game metaphors, design principles, and technologies can be utilized for alternative content and context delivery. The focus is on the next generation Internet and beyond.

The approach combines theory and practice, art and science, education and entertainment, to create an environment that supports diverse forms of expression in a wide range of applications.

The methods include sampling, misuse, hacking, appropriation, reverse engineering, and custom creation in the interest of open-source innovation and critical intervention.

Since its inception in 1999, the Game Lab has been physically housed in the Claire Trevor School of the Arts (SOTA). In 2002 we established a co-located facility in The California Institute for Telecommunication and Information Technology (Calit2).

A Composed Open Architecture Software System at Run-Time

Calendars Monday 26 Apr 2010

1:00pm Review proposal

3:00pm Work on JSS paper draft

Tasks: Proposal review

```
dmraid          initctl              lvrename             pam_tally2                   rpcbind      vgexport
dmraid.static   initlog              lvresize             pam_timestamp_check          rpc.statd    vgextend
dmsetup         insmod               lvs                  parted                       rrestore     vgimport
dosfsck         insmod.static        lvscan               partprobe                    rsyslogd     vgmerge
dosfslabel      install-info         MAKEDEV              partx                        rtmon        vgmknodes
dump            installkernel        mdadm                pccardctl                    runlevel     vgreduce
dumpe2fs        ip                   mdadm.static         pcmcia-check-broken-cis      runuser      vgremove
e2fsck          ip6tables            mdassemble           pcmcia-socket-startup        scsi_id      vgrename
e2image         ip6tables-multi      mdassemble.static    pidof                        securetty    vgs
e2label         ip6tables-restore    microcode_ctl        pivot_root                   service      vgscan
e2undo          ip6tables-save       mii-diag             plipconfig                   setfiles     vgsplit
ether-wake      ipmaddr              mii-tool             plymouthd                    setpci       weak-modules
ethtool         iptables             mingetty             portrelease                  setsysfont   ypbind
extlinux        iptables-multi       mkdosfs              portreserve                  sfdisk
fdisk           iptables-restore     mke2fs               poweroff                     shutdown
findfs          iptables-save        mkfs                 pppoe                        slattach
[liveuser@localhost sbin]$ ls ../selinux
access    checkreqprot          compat_net  deny_unknown  initial_contexts  mls                   policyvers      user
avc       class                 context     disable       load              null                  reject_unknown
booleans  commit_pending_bools  create      enforce       member            policy_capabilities   relabel
[liveuser@localhost sbin]$
```

Fig. 9. A third instantiation at run-time (Opera, Google Docs, Gnome Evolution, Fedora) determining yet another niche conforming to the software supply network of Fig. 4.

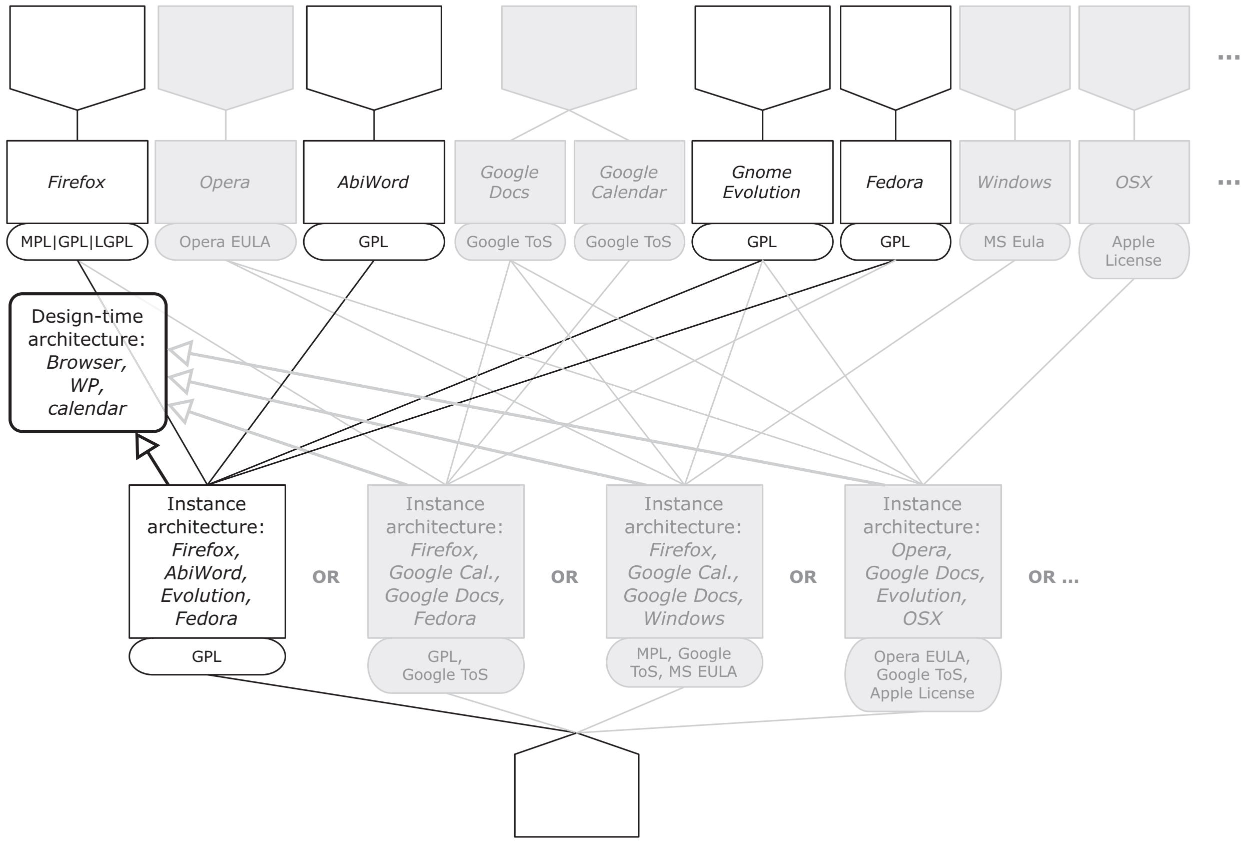

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
...
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

Fig. 10. The ecosystem niche for one instance architecture.

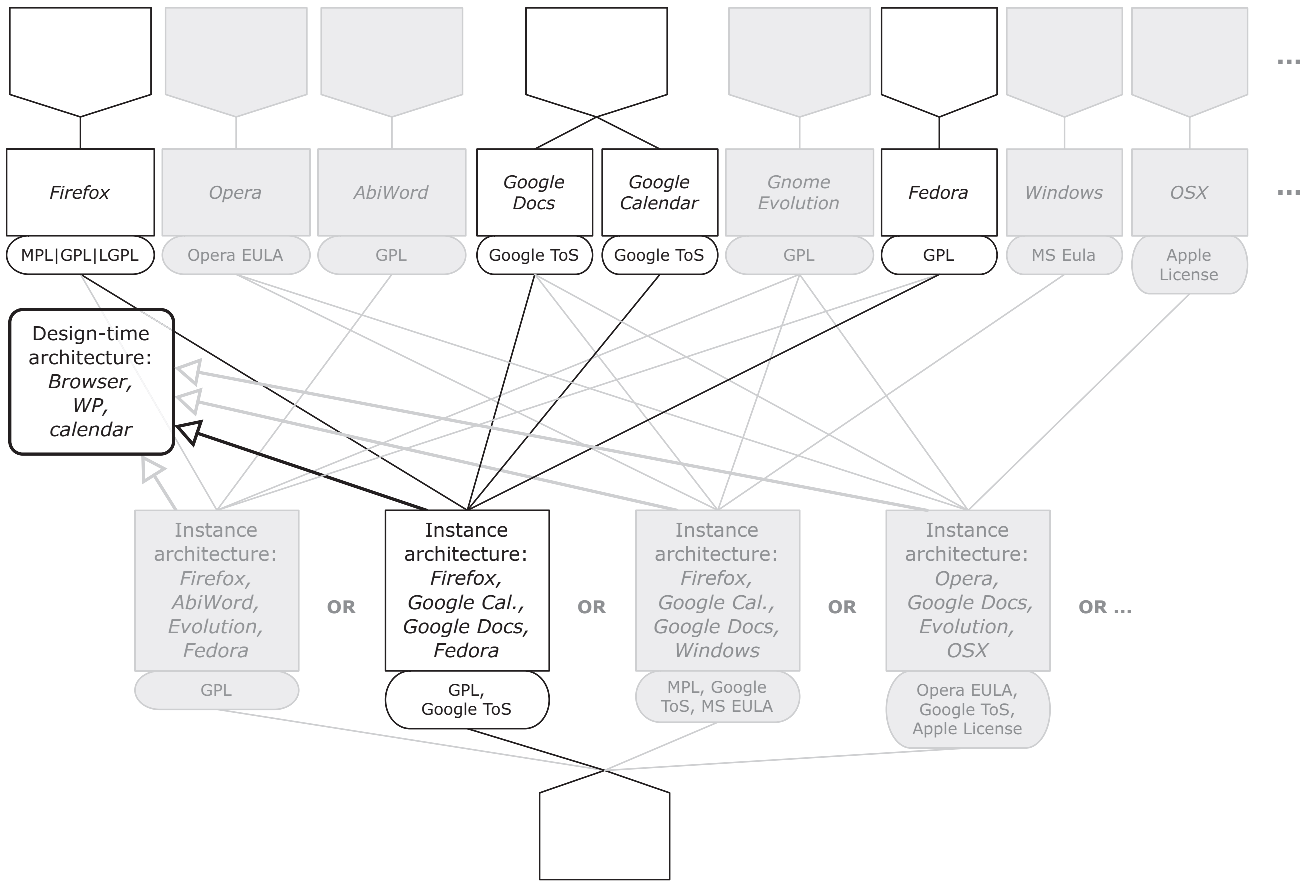

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

Fig. 11. The ecosystem niche for a second instance architecture.

- It must distinguish the architectural constructs significant for software licenses, and embody their effects on rights and obligations (Section 4).
- It must define the system's *license architecture*, the abstraction of its software architecture annotated with licenses, connector types, etc. that determines the system's *virtual license* (overall rights and obligations) and from which the virtual license can be calculated (Alspaugh et al., 2009b, 2011).
- It must account for alternative ways in which software systems, components, and licenses can evolve (Section 6).
- It must provide an automated environment for creating and managing license architectures. We have developed a prototype that manages the license architecture as a view of the system architecture (Alspaugh et al., 2009b, 2011).

6. 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. For the application of these mechanisms to systems rather than ecosystems, see our previous work (Alspaugh et al., 2009a,b, 2011, 2010; Scacchi and Alspaugh, 2008).

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 Word). However, changes in the format or structure of component APIs may necessitate build-time and run-time updates to component connectors. Fig. 12 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 by changing connectors between components rearranging connectors in a different configuration, or changing the interface through which a connector accesses a component, altering the system characteristics. 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. An example is the replacement of word processing, calendaring, email components, and connectors to them 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 remotely from 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 preferences for kinds of licenses and the consequences of 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

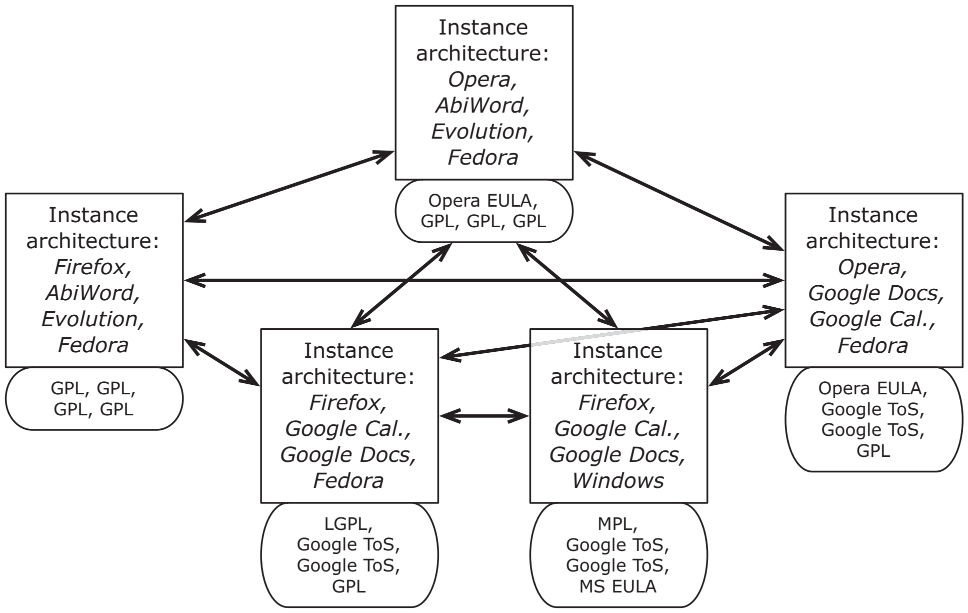

Instance architecture: Opera, AbiWord, Evolution, Fedora
Opera EULA, GPL, GPL, GPL
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL, GPL, GPL, GPL
Instance architecture: Opera, Google Docs, Google Cal., Fedora
Opera EULA, Google ToS, Google ToS, GPL
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
LGPL, Google ToS, Google ToS, GPL
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, Google ToS, MS EULA

Fig. 12. Possible evolutionary paths among a few instance architectures; some paths are impractical due to the changes in license obligations.

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 Fig. 12 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.

In response to different 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. Fig. 13 shows an array of choices among types of licenses for different types of components that appear in the OA example system. Each choice determines the obligations that component producers can demand of their consumers in exchange for the access/usage rights they offer.

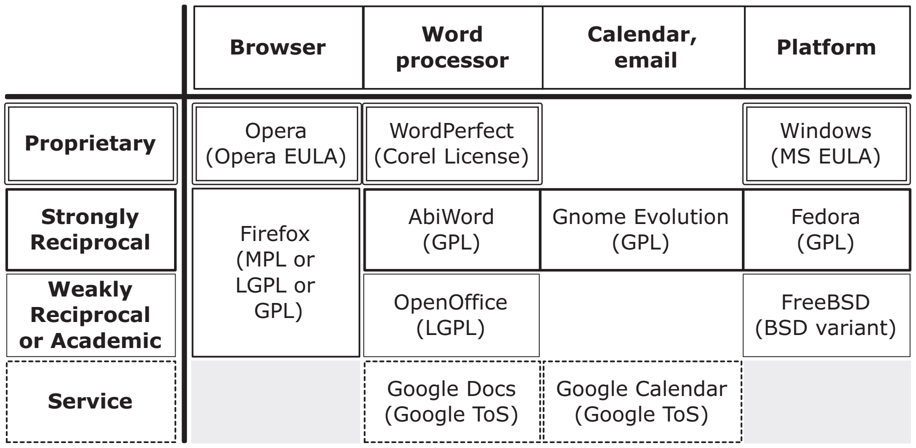
The interdependence of producers, integrators, and consumers results in a co-evolution of software systems and social networks within an OA ecosystem (Scacchi, 2007). Closely coupled components from different producers must evolve in parallel in order for each to provide its services, as evolution in one will typically require a matching evolution in the other. Producers may manage their evolution with a loose coordination among releases, as for example is done between the Gnome and Mozilla organizations. Each release of a producer component creates a tension 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 (Alspaugh et al., 2009b), 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.

7. 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 (Bosch, 2000; Clements and Northrop, 2001). 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



	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)	

Fig. 13. Some architecture choices and their license categories.

architecture depicted in Fig. 5, the build-time architecture in Fig. 6, and the instantiated run-time architectures in Figs. 7–9.

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 (Noll and Scacchi, 1999) that spans multiple independent OSS projects and software production enterprises; virtual in the sense that both the enterprise and the SPL are emergent phenomena rather than intended and embodied by existing organizations and business plans. SPL concepts are thus compatible with OA systems that are composed from heterogeneously licensed components, and 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 (2009) 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, Microsoft Windows 7 OS phones, Intel MeeGo/Tizen 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, mobile, 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 or for 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 example system shown in Fig. 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 Fig. 12 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.

8. 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 in which a system resides. Understanding and describing software ecosystem niches is a key contribution of this work. An example system of an open architecture software system that articulates different software supply networks as ecosystem 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 exist through the association between open architectures, software component licenses, and software ecosystems.

Finally, in previous work (Alspaugh et al., 2010, 2009b,c) 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 remains 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 grants #N00244-10-1-0038 and #N00244-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School. No review, approval, or endorsement is implied.

Appendix: Kinds of 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 *permissive* or *reciprocal*. Permissive 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 permissive 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 GPL and the Affero GPL (AGPL). The distinctive purpose of GPL is to increase the commons of OSS, by requiring software incorporating GPL'd components to be released only under GPL (for various definitions of "incorporating"). AGPL additionally prevents software components licensed under it 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 permissive 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.

Most license provisions have focused on copyright issues and the rights to reproduce, prepare derivative works. and distribute copies that are governed by copyright law. Newer licenses often cover patent issues as well and the rights to make, use, sell or offer for sale, and import that are governed by patent law. These licenses either grant a restricted patent license or explicitly exclude the granting of patent rights. However, some important licenses are constructed so that some or all rights under the license terminate if the licensee institutes patent infringement suits related to the licensed software (specifics vary by license), for example Apache 2.0 and MPL 1.1. Proprietary licenses often place limits on the use of the licensed software as part of the contractual obligations imposed by the license, whether the software is patented or not.

Both proprietary and OSS licenses typically disclaim liability, assert no warranty is implied, and (for reasons based in trademark law and beyond the scope of this work) obligate licensees to not use the licensor's name or trademarks.

The Open Source Initiative (OSI) maintains a widely respected definition of "open source" and gives its approval to licenses that meet it (OSI, 2011). 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 permissive 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 (Jensen and Scacchi, 2011). 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.

HLS designers have developed a number heuristics to guide architectural design while avoiding some license conflicts. First, it may be 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.

References

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

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

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

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

Alspaugh, T.A., Asuncion, H.U., Scacchi, W., 2011. Presenting software license conflicts through argumentation. In: 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011), pp. 509–514.

Alspaugh, T.A., Scacchi, W., Asuncion, H.U., 2010. Software licenses in context: the challenge of heterogeneously-licensed systems. Journal of the Association for Information Systems 11 (11), 730–755.

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

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

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

Bosch, J., Bosch-Sijtsema, P., 2010. From integration to composition: on the impact of software product lines, global development and ecosystems. Journal of Systems and Software 83 (1), 67–76.

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

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.

Chromium, 2011. Chromium terms and conditions. http://code.google.com/chromium/terms.html.

Chromium issues, 2009. Issue 10638: remove JSCRE from about:credits. https://code.google.com/p/chromium/issues/detail?id=10638.

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

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

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.

Hutchinson, G.E., 1957. Concluding remarks. Cold Spring Harbor Symposia on Quantitative Biology 22 (2), 415–427.

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

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

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

Jensen, C., Scacchi, W., 2011. License update and migration processes in open source software projects. In: Hissam, S., Russo, B., de Mendonça Neto, M., Kon, F. (Eds.), Open Source Systems: Grounding Research. IFIP Advances in Information and Communication Technology, pp. 177–195.

Kuehnel, A.-K., 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.

Kuhl, F., Weatherly, R., Dahmann, J., 1999. Creating Computer Simulation Systems: An Introduction to the High Level Architecture. Prentice Hall.

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

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

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.

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

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

Open Source Initiative, 2011. Open Source Definition. http://www.opensource.org/docs/osd.

Oreizy, P., 2000. Open Architecture Software: A Flexible Approach to Decentralized Software Evolution. PhD Thesis, University of California, Irvine.

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.

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

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.

Scacchi, W., Alspaugh, T.A., 2008. Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In: 5th Annual Acquisition Research Symposium, pp. 230–214.

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

Unity Technologies, December 2008. End User License Agreement. http://unity3d.com/unity/unity-end-user-license-2.x.html.

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.

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.

Walt Scacchi is a senior research scientist and research faculty member at the Institute for Software Research, University of California, Irvine. He received a Ph.D. in Information and Computer Science from UC Irvine in 1981. From 1981–1998, he was on the faculty at the University of Southern California. In 1999, he joined the Institute for Software Research at UC Irvine. He has published more than 150 research papers, and has directed 60 externally funded research projects. Last, in 2012, he serves as General Co-Chair of the 8th. IFIP International Conference on Open Source Systems (OSS2012).

Thomas Alspaugh is a project scientist at the Institute for Software Research, University of California, Irvine, where he was on the faculty from 2002–2008. From 2008–2011 he served as an adjunct faculty member in Computer Science at Georgetown University. His research interests are in software engineering, requirements, and licensing. Before completing his Ph.D. in 2002 at North Carolina State University, he worked as a software developer, team lead, and manager in industry, and as a computer scientist at the Naval Research Laboratory on the Software Cost Reduction (A-7) project.