*First International Workshop on Software Ecosystems (IWSECO-2009),* pp. 4–18

# The Role of Software Licenses in Open Architecture Ecosystems

Thomas A. Alspaugh, Hazeline U. Asuncion, and Walt Scacchi

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
{alspaugh,hasuncion,wscacchi}@ics.uci.edu

**Abstract.** The role of software ecosystems in the development and evolution of open architecture systems has received insufficient consideration. Such systems are composed of heterogeneously-licensed components, open source or proprietary or both, in an architecture in which evolution can occur by evolving existing components or by replacing them. The software licenses of the components both facilitate and constrain the system's ecosystem, and the rights and duties of the licenses are crucial in producing an acceptable system. We discuss software ecosystems of open architecture systems from the perspective of an architect or an acquisition organization, and outline how our automated tool and environment help address their challenges, support reuse, and assist in managing coevolution and component interdependence.

## 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) [1], 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, the organization becomes an integrator of components largely produced elsewhere, connected with 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 component producers, and in another direction by the needs of its 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*, 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. We find that an OA software ecosystem involves organizations and individuals producing and consuming components, and supply paths from producer to consumer; 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.

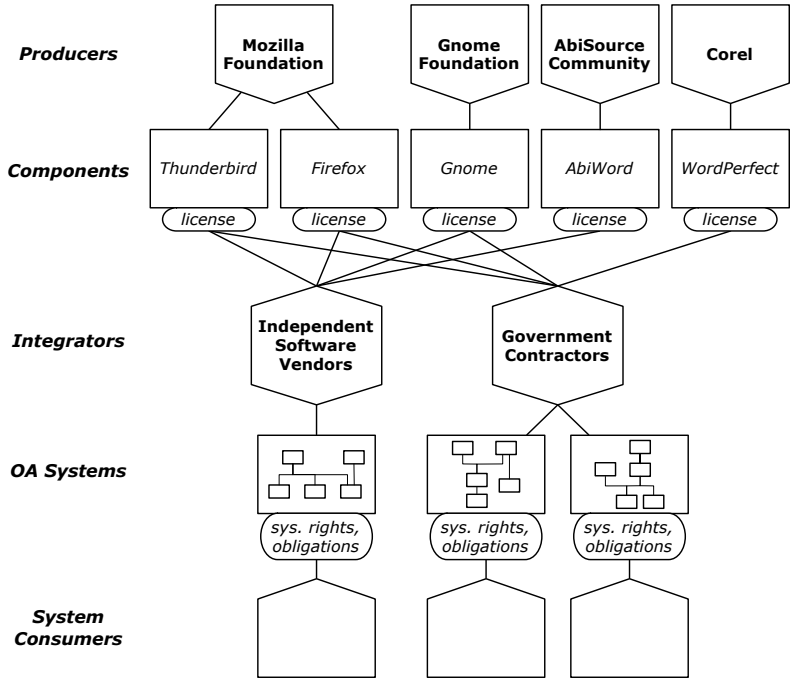


**Fig. 1.** A hypothetical ecosystem in which OA systems are developed

In order to most effectively use an OA approach in developing and evolving a system, it is essential to consider this OA ecosystem. An OA system draws on components from proprietary vendors and open source projects. Its architecture is made possible by the existing general ecosystem of producers, from which the initial components are chosen. The choice of a specific OA begins a specialized software ecosystem involving components that meet (or can be shimmed to meet) the open interfaces used in the architecture. We do not claim

this is the best or the only way to reuse components or produce systems, but it is an ever more widespread way. In this paper we build on previous work on heterogeneously-licensed systems [2–4] by examining how OA development affects and is affected by software ecosystems, and the role of component licenses in OA software ecosystems.

A motivating example of this approach is the Unity game development tool, produced by Unity Technologies [5]. 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.

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.

***By component replacement***— One or more components may be replaced by others with different behaviours but the same interface, or with a different interface and the addition of shim code to make it match.

***By architecture evolution***— The OA can evolve, using the same components but in a different configuration, altering the system's characteristics. For example, as discussed in Section 4, changing the configuration in which a component is connected can change how its license affects the rights and obligations for the overall system.

***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.

***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.

The interdependence of integrators and producers results in a co-evolution of software within an OA ecosystem. 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 through the ecosystem relationships with consumers and their releases of OA systems using those components, as integrators accomodate the choices of available, supported components with their own goals and needs. As discussed in our previous work [4], license rights and obligations are manifested at each component's interface, then mediated through the system's OA to entail the rights and corresponding obligations for the system as a whole. As a result, integrators must frequently re-evaluate an OA system's 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.

We propose that such support must have several characteristics.

- It must rest on a license structure of rights and obligations (Section 5), focusing on obligations that are enactable and testable. 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 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 take account of the distinctions between the design-time, build-time, and distribution-time architectures (Sections 4, 5, 6) and the rights and obligations that come into play for each of them.
- It must distinguish the architectural constructs significant for software licenses, and embody their effects on rights and obligations (Section 4).
- It must define license architectures (Section 6).

– 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 its system architecture [4].
– Finally, it must automate calculations on system rights and obligations so that they may be done easily and frequently, whenever any of the factors affecting rights and obligations may have changed (Section 7).

In the remainder of this paper, we survey some related work (Section 2), provide an overview of OSS licenses and projects (Section 3), define and discuss characteristics of open architectures (Section 4), introduce a structure for licenses (Section 5), outline license architectures (Section 6), sketch our approach for license analysis (Section 7), and conclude (Section 8).

## 2 Related Work

Jansen *et al.* discuss the perspective of software vendors on software ecosystems [6]. Scacchi examines how free/open source software projects become part of a multi-project ecosystem, interdependent in the context of evolution and reuse [7]. The present work examines the point of view of organizations that develop or acquire OA systems.

Brown and Booch discuss issues that arise in the reuse of OSS components, such as that interdependence causes changes to propagate, and versions of the components evolve asynchronously giving rise to co-evolution of interrelated code in the OA [8]. If the components evolve, the OA system itself is evolving. The evolution can also include changes to the licenses, and the licenses can change from version to version.

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 [9].

Scacchi and Alspaugh examine the features of software architecture and OSS licenses that affect the success of an OA strategy [3].

There are a number of discussions of OSS licenses, such as Rosen [10] and Fontana *et al.* [11].

## 3 Open-Source Software (OSS)

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. The goals 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. Example reciprocal licenses are GPL, the Mozilla Public License (MPL), and the Common Public License.

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 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 [12]. OSI maintains and publishes a repository of approximately 70 approved OSS licenses.

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 homogenously 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.

## 4 Open Architecture (OA)

Open architecture (OA) software is a customization technique introduced by Oreizy [1] 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. [5]). Using this approach can lower development costs and increase reliability and function [3]. 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 [3, 13].

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed [14].

**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" [16]. Their source code is available and they can be rebuilt. Each may have its own distinct license.

**Executable components**—These components are in binary form, and the source code may not be open for access, review, modification, or possible redistribution [10]. 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" [17]. APIs are not and cannot be licensed, and 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 [18], 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 connection**—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. Methods of connection 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.

Figure 2 shows a high-level view of a reference architecture that includes all the kinds of software elements listed above. This reference architecture has

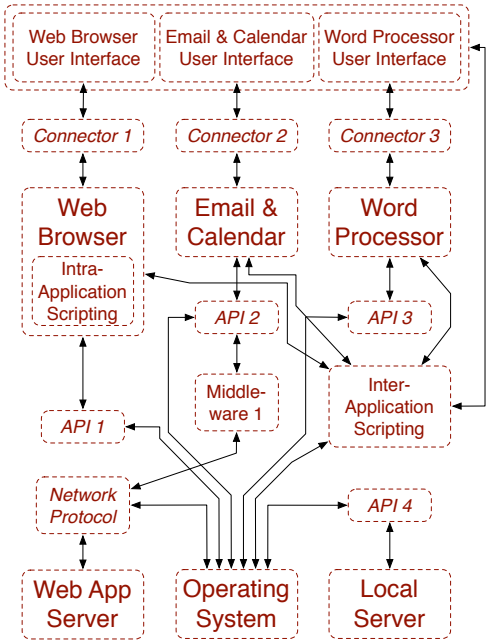


**Fig. 2.** Reference architecture for a heterogeneously-licensed e-business system; connectors (which have no license) are italicized

been instantiated in a number of configured systems that combine OSS and closed source components. One such system handles time sheets and payroll at our university; another implements the web portal for a university research lab (`http://proxy.arts.uci.edu/gamelab/`). The configured systems consist of software components such as a Mozilla Web browser, Gnome Evolution email client, and WordPerfect word processor, all running on a Linux operating system accessing file, print, and other remote networked servers such as an Apache Web server. Components are interconnected through a set of software connectors that bridge the interfaces of components and combine the provided functionality into the system's services.

## 5 Software Licenses

Copyright law is the common basis for software licenses, and gives the original author of a work certain exclusive rights: the rights to use, copy, modify, merge, publish, distribute, sub-license, and sell copies. The author may license these rights, individually or in groups, to others; the license may give a right either exclusively or non-exclusively. After a period of years, copyright rights enter the public domain. Until then copyright may only be obtained through licensing.

Licenses typically impose obligations that must be met in order for the licensee to realize the assigned rights. Common obligations include the obligation

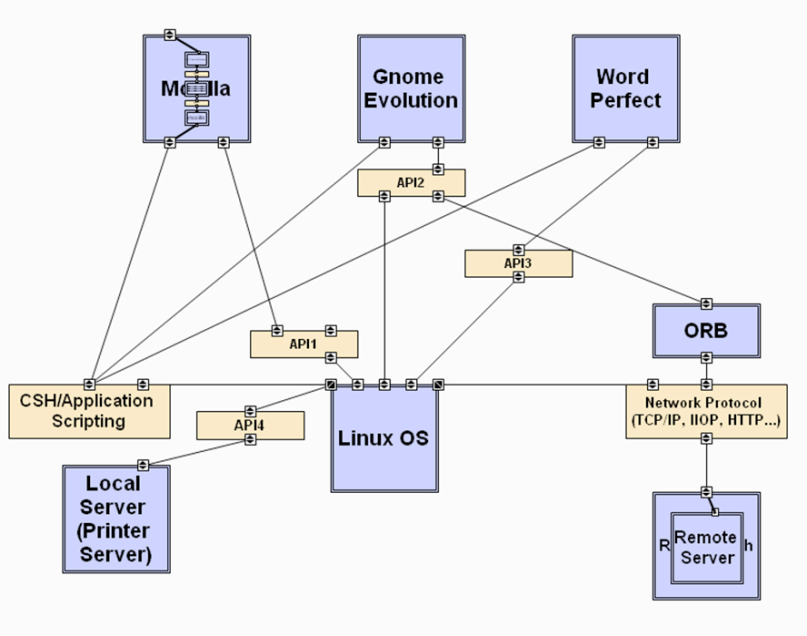


**Fig. 3.** Instance architecture for a heterogeneously-licensed e-business system (some components show high-level internal structure)

to publish at no cost any source code you modify (MPL) or the reciprocal obligation to publish all source code included at build-time or statically linked (GPL). The obligations may conflict, as when a GPL'd component's reciprocal obligation to publish source code of other components is combined with a proprietary component's license prohibition of publishing its source code. In this case, no rights may be available for the system as a whole, not even the right of use, because the two obligations cannot simultaneously be met and thus neither component can be used as part of the system.

The basic relationship between software license rights and obligations can be summarized as follows: if the specified obligations are met, then the corresponding rights are granted. For example, if you publish your modified source code and sub-licensed derived works under MPL, then you get all the MPL rights for both the original and the modified code. However, license details are complex, subtle, and difficult to comprehend and track—it is easy to become confused or make mistakes. The challenge is multiplied when dealing with configured system architectures that compose a large number of components with heterogeneous licenses, so that the need for legal counsel begins to seem inevitable [10, 11].

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

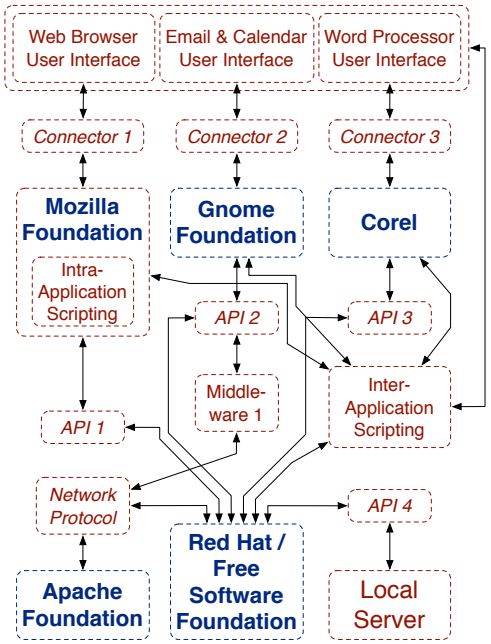


**Fig. 4.** Reference architecture components supplied by an organization, indicating the integrator's dependencies on suppliers, mediated by interfaces and licenses

eight fundamental jural relations [19], of which we use *right*, *duty*, *no-right*, and *privilege*. 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 5 displays the tuples and associations for two of the rights and their associated obligations for the academic BSD software license. Note that the first right is granted without corresponding obligations.

Heterogeneously-licensed system designers have developed a number heuristics to guide architectural design while avoiding some license conflicts. First, it is possible to use a reciprocally-licenced 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 a Limited General Public License connector), or run-time plug-ins. A second approach

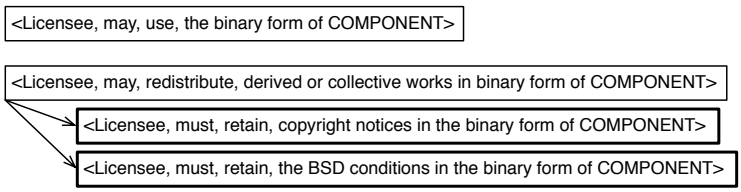


**Fig. 5.** Tuples for some rights and obligations of the BSD license

used by a number of large organizations is simply to avoid using any reciprocally-licensed components. 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. Thus, automated support is needed to manage the multi-component, multi-license complexity.

## 6 License Architectures

Our license model forms a basis for effective reasoning about licenses in the context of actual systems, and calculating the resulting rights and obligations. In order to do so, we need a certain amount of information about the system's configuration at design-, build-, distribution-, and run-time. The needed information comprises the *license architecture*, an abstraction of the system architecture:

1. the set of components of the system;
2. the relation mapping each component to its license;
3. the relation mapping each component to its set of sources; and
4. the relation from each component to the set of components in the same license scope, for each license for which "scope" is defined (e.g. GPL), and from each source to the set of sources of components in the scope of its component.

With this information and definitions of the licenses involved, we can calculate rights and obligations for individual components or for the entire system, and guide heterogeneously-licensed system design.

## 7 License Analysis

Given a specification of a software system's architecture, we can associate software license attributes with the system's components, connectors, and subsystem architectures, resulting in a license architecture for the system, and calculate the copyright rights and obligations for the system's configuration. Due

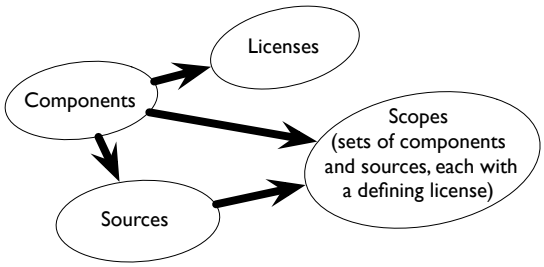


**Fig. 6.** The license architecture metamodel

to the complexity of license architecture analysis, and the need to re-analyze every time a component evolves, a component's license changes, a component is substituted, or the system architecture changes, OA integrators really need an automated license architecture analysis environment. We are developing a prototype of such an environment [4].

We use an architectural description language specified in xADL [20] to describe OAs that can be designed and analyzed with a software architecture design environment [21], such as ArchStudio4 [22]. We have built the Software Architecture License Analysis module on top of ArchStudio's Traceability View [23]. This allows for the specification of licenses as a list of attributes (license tuples) using a form-based user interface in ArchStudio4 [21, 22].

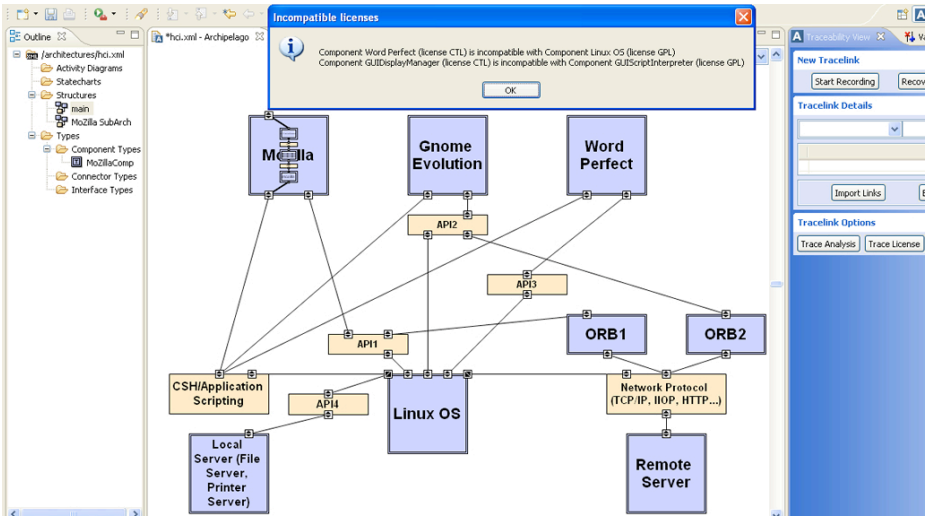


**Fig. 7.** Automated tool performing license analysis of instance architecture (version information not shown)

We analyze rights and obligations as described below [4].

### 7.1 Propagation of reciprocal obligations

We follow the widely-accepted interpretation that build-time static linkage propagate the reciprocal obligations, but appropriate license firewalls do not. Analysis begins, therefore, by propagating these obligations along all connectors that are not license firewalls.

### 7.2 Obligation conflicts

An obligation can conflict with another obligation, or with the set of available rights, by requiring a copyright right that has not been granted. For instance, a proprietary license may require that a licensee must not redistribute source code, but GPL states that a licensee must redistribute source code. Thus, the conflict appears in the modality of the two otherwise identical obligations, "must not" in the proprietary license and "must" in GPL.

### 7.3 Rights and obligations calculations

The rights available for the entire system (use, copy, modify, etc.) are calculated as the intersection of the sets of rights available for each component of the system. If a conflict is found involving the obligations and rights of linked components, it is possible for the system architect to consider an alternative linking scheme, e.g. using one or more connectors along the paths between the components that act as a license firewall. This means that the architecture and the automated environment together can determine what OA design best meets the problem at hand with available software components. Components with conflicting licenses do not need to be arbitrarily excluded, but instead may expand the range of possible architectural alternatives if the architect seeks such flexibility and choice.

## 8 Conclusion

This paper discusses the role of ecosystems in the development and evolution of OA systems. License rights and obligations play a key role in how and why an OA system evolves in its ecosystem. We note that license changes across versions of components is a characteristic of OA systems and ecosystems. A structure for modeling software licenses and the license architecture of a system and automated support for calculating its rights and obligations are needed in order to manage a system's evolution in the context of its ecosystem. We have outlined an approach for achieving these and sketched how they further the goal of reusing components in developing software-intensive systems. Much more work remains to be done, but we believe this approach turns a vexing problem into 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 endorsement implied.

## References

1. Oreizy, P.: Open Architecture Software: A Flexible Approach to Decentralized Software Evolution. PhD thesis, University of California, Irvine (2000)
2. German, D.M., Hassan, A.E.: License integration patterns: Dealing with licenses mismatches in component-based development. In: 28th International Conference on Software Engineering (ICSE '09). (May 2009)
3. Scacchi, W., Alspaugh, T.A.: Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In: 5th Annual Acquisition Research Symposium. (May 2008)
4. Alspaugh, T.A., Asuncion, H.U., Scacchi, W.: Analyzing software licenses in open architecture software systems. In: 2nd International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS). (May 2009)
5. Unity Technologies: End User License Agreement (December 2008) `http://unity3d.com/unity/unity-end-user-license-2.x.html`.
6. Jansen, S., Finkelstein, A., Brinkkemper, S.: A sense of community: A research agenda for software ecosystems. In: ICSE Companion '09: Companion of the 31st International Conference on Software Engineering. (May 2009) 187 190
7. Scacchi, W.: Free/open source software development. In: ESEC/FSE 2007: 6th Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering. (September 2007) 459–468
8. Brown, A.W., Booch, G.: Reusing open-source software and practices: The impact of open-source on commercial vendors. In: Software Reuse: Methods, Techniques, and Tools (ICSR-7). (April 2002)
9. Ven, K., Mannaert, H.: Challenges and strategies in the use of open source software by independent software vendors. Information and Software Technology **50**(9-10) (2008) 991–1002
10. Rosen, L.: Open Source Licensing: Software Freedom and Intellectual Property Law. Prentice Hall (2005)
11. Fontana, R., Kuhn, B.M., Moglen, E., Norwood, M., Ravicher, D.B., Sandler, K., Vasile, J., Williamson, A.: A Legal Issues Primer for Open Source and Free Software Projects. Software Freedom Law Center (2008)
12. Open Source Initiative: Open Source Definition (2008) `http://www.opensource.org/`.
13. Alspaugh, T.A., Antón, A.I.: Scenario support for effective requirements. Information and Software Technology **50**(3) (February 2008) 198–220
14. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
15. Feldt, K.: Programming Firefox: Building Rich Internet Applications with XUL. O'Reilly Media, Inc. (2007)
16. Nelson, L., Churchill, E.F.: Repurposing: Techniques for reuse and integration of interactive systems. In: International Conference on Information Reuse and Integration (IRI-08). (2006) 490

17. Meyers, B.C., Oberndorf, P.: Managing Software Acquisition: Open Systems and COTS Products. Addison-Wesley Professional (2001)
18. Kuhl, F., Weatherly, R., Dahmann, J.: Creating computer simulation systems: an introduction to the high level architecture. Prentice Hall (1999)
19. Hohfeld, W.N.: Some fundamental legal conceptions as applied in judicial reasoning. Yale Law Journal **23**(1) (November 1913) 16–59
20. Institute for Software Research: xADL 2.0. Technical report, University of California, Irvine (2009) `http://www.isr.uci.edu/projects/xarchuci/`.
21. Medvidovic, N., Rosenblum, D.S., Taylor, R.N.: A language and environment for architecture-based software development and evolution. In: ICSE '99: Proceedings of the 21st international Conference on Software Engineering. (1999) 44–53
22. Institute for Software Research: ArchStudio 4. Technical report, University of California, Irvine (2006) `http://www.isr.uci.edu/projects/archstudio/`.
23. Asuncion, H., Taylor, R.N.: Capturing custom link semantics among heterogeneous artifacts and tools. In: 5th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE). (May 2009)