UCI-AM-10-021

ACQUISITION RESEARCH
SPONSORED REPORT SERIES

Investigating the Acquisition of Software Systems that Rely on Open Architecture and Open Source Software

March 2010

by

Dr. Walt Scacchi, Senior Research Scientist,

Thomas A. Alspaugh, Assistant Professor and

Hazel Asuncion, Post-doctoral Researcher

Institute for Software Research

University of California, Irvine

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

The research presented in this report was supported by the Acquisition Chair of the Graduate School of Business & Public Policy at the Naval Postgraduate School.

Executive Summary

In 2007-08, we began an investigation of problems, issues, and opportunities that arise during the acquisition of software systems that rely on open architectures and open source software. The current effort funded for 2009 seeks to continue and build on the results in this area, while refining its focus to center on the essential constraints and tradeoffs we have identified for software-intensive systems with open architecture (OA) and continuously evolving open source software (OSS) elements. The U.S. Air Force, Army, and Navy have all committed to an open technology development strategy that encourages the acquisition of software systems whose requirements include the development or composition of an OA for such systems, and the use of OSS systems, components, or development processes when appropriate. Our goal is to further develop and document foundations for emerging policy and guidance for acquiring software systems that require OA and that incorporate OSS elements. This report documents and describes the findings and results that we have produced as a result of our research into the area of the acquisition of software systems that rely on OA and OSS. In particular, it includes four research papers that have been refereed, reviewed, presented, and published in national and international research conferences, symposia, and workshops.

Research Description

OSS is an integrated web of people, processes, and organizations, including project teams operating as virtual organizations [Scacchi 2002, 2007]. There is a basic need to understand how to identify an optimal mix of OSS within Open Architectures (OA) as products, production processes, practices, community activities, and multi-project (or multi-organization) software ecosystem. However, the relationship among OA, OSS, requirements, and acquisition is poorly understood [cf. Scacchi 2002, Naegle and Petross 2007]. Subsequently, in 2007-08, we began by examining how different OSS licenses can encumber software systems within OA, which therefore give rise to new requirements for how best to acquire software-intensive systems with OA and OSS elements [Scacchi and Alspaugh 2008].

Across the three military services within the DoD, OA means different things and is seen as the basis for realizing different kinds of outcomes [Justice 2007, Navy 2007, Riechers 2007]. Thus, it is unclear whether the acquisition of a software system that is required to incorporate an OA, as well as utilize OSS technology and development processes [Wheeler 2007], for one military service will realize the same kinds of benefits anticipated for OA-based systems by another service. Somehow, DoD acquisition program managers must make sense of or reconcile such differences in expectations and outcomes from OA strategies in each service and across DoD. Yet there is little explicit guidance or reliance on systematic empirical studies for how best to develop, deploy, and sustain complex software-intensive military systems in the different OA and OSS presentations and documents that have so far been disseminated [Starrett 2007, Weathersby 2007, Wheeler 2007].

It is becoming clear that verification and validation (V&V) is a crucial activity for OSS in OA. Two key benefits of OSS are its reliability and its openness to rapid, agile evolution in response to changing needs. Following Justice [2007], we envision that warfighters will be not only users of OSS but also contributing “developers” to it, as they know what is needed "right now" to give them the edge over their opponents and are best placed to translate that quickly into new system capabilities. However, it is still important that OSS/OA systems be reliable and remain open. The benefit of having a new feature to respond to a current threat is reduced if the change causes a needed existing feature to stop working, or in the worst case causes the system to fail unexpectedly. With development activities extended out close to "the tip of the spear" it is also necessary to extend V&V out to the same developers and their quick-response development. Not only do they need to be able to quickly validate the changes they have made, they also need to be able to do quick, highly automated regression testing to identify any existing functions that the new changes have interfered with. At the same time, DoD needs a structure for managing the evolution of OSS and OA systems at higher levels, to deal with the decisions of which "spear-tip" changes to fold into the system for larger groups of users, and when, and to resolve the inevitable conflicts that will arise as different groups of developers take the system in different, sometimes conflicting directions. In addition, verification must confirm that the changed system remains open, as well as correct. V&V must be done quickly and convincingly at these broader levels as well, and will be important in supporting decisions about which modifications to disseminate when. The recent work Berzins [2008] presented at the 2008 Acquisition Research Symposium shows one approach that applies a high degree of automation in order to make regression testing more efficient and more manageable in response to system modifications.

We now turn to address the problems for acquisition research in this area.

Problem for Acquisition Research

OA seems to imply software system architectures incorporating OSS components and open application program interfaces (APIs). But not all software system architectures incorporating OSS components and open APIs will produce OA [cf. Scacchi and Alspaugh 2008], since OA depends on: (a) how/why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, (c) whether the copyright licenses assigned to different OSS components encumber all/part of a software system's architecture into which they are integrated. Similarly, (d) alternative architectural configurations and APIs for a given system may or may not produce an OA. Subsequently, we believe this can lead to situations in which if program acquisition stipulates a software-intensive system with an OA and OSS, and the architectural design of a system constrains system requirements (i.e., what requirements can be satisfied by a given system architecture, or given system requirements what architecture is implied), then the resulting software system may or may not embody an OA.

OSS processes encourage users (here, warfighters) to become contributing developers, and in an OA context this entails distributing V&V to them as well. A traditional approach to system and component requirements may or may not support OSS components' rapid and fluid evolution, and the distributed, efficient V&V that will be needed at all levels from system down to components. The need to manage evolution at a high level also imposes constraints on requirements and, quite probably, architectures.

Thus, given the goal of realizing an OA strategy, together with the use of evolving OSS components and open APIs, *how should program acquisition, system requirements, software V&V, open architectures, and post-deployment system support be aligned to achieve this goal?* As such, this is the central research problem we are investigating in order to identify principles, best practices, and knowledge for how best to insure the success of the OA strategy when OSS and open APIs are required or otherwise employed. Without such knowledge, program acquisition managers and PEOs are unlikely to acquire software-intensive systems that will result in an OA that is clean, robust and transparent. This may frustrate the ability of program managers or PEOs to realize faster, better, and cheaper software acquisition, development, and post-deployment support.

Issues for Acquisition Research

Based on current research into the acquisition of OA systems with OSS components [Scacchi and Alspaugh 2008], this research project investigated the following kinds of research questions: How does the interaction of requirements and architectures for OA systems incorporating OSS components facilitate or inhibit acquisition practices over time? What are the best available ways and means for continuously verifying and validating the functionality, correctness, and openness of OA when OSS components are employed? How do OA systems evolve over time when incorporating continuously improving OSS components? How can use of continuously evolving OSS in OA be combined with the need to verify and validate critical systems and to manage their evolution? How do reliability and predictability trade-off against the cost and flexibility of an OA system when incorporating OSS components? How should OA software systems be developed and deployed to support warfighter modification in the field or participation in post-deployment system support, when OSS components are employed?

Prospects for longer-term Acquisition-related research

Each of the military services has committed to orienting their major system acquisition programs around the adoption of an OA strategy that in turn embraces and encourages the adoption, development, use, and evolution of OSS. Thus, it would seem there is a significant need for sustained research that investigates the interplay and inter-relationships between (a) current/emerging guidelines for the acquisition of software-intensive systems within the DoD community, and (b) how software systems that employ an OA incorporating OSS products and production processes are essential to improving the effectiveness of future, software-intensive program acquisition efforts.

Findings and Results

Based on the research studies conducted during this project during the 2009 project year, we produced a series of four papers that have been reviewed and refereed by software engineering, open source software, open architecture, and acquisition researchers. Each of these papers has been jointly authored by the members of our project team, Walt Scacchi, Thomas Alspaugh, and Hazel Asuncion, and each has been presented at top-tier international research conferences, national symposia, or workshops, and all have been published. Thus, we are reasonably confident about the quality and nature of our findings, as well as to the veracity of the research approach we have pursued, and the computational tools and analytical methods we have employed along the way. Furthermore, we are continuing to build on these results during the next annual cycle of research beginning in 2010.

Our first paper, "Software Licenses, Open Source Components, and Open Architectures," appears in *Proc. 6th. Annual Acquisition Research Symposium*, Monterey, CA, May 2009. This paper lays the foundation for our approach and direction for problems to investigate, as are documented in the remaining three papers. It was presented and well received at the ARS, and received many favorable comments from members of the Acquisition community in attendance.

Our second paper, "Analyzing Software Licenses in Open Architecture Software Systems," *Proc. Workshop on Emerging Trends in FLOSS Research and Development, Intern. Conf. Software Engineering*, Vancouver, Canada, May 2009. This paper represents findings that are framed to specifically address the relationship between OA, OSS, and software licenses, which subsequently became part of the core of the doctoral dissertation of our project team member, Hazel Asuncion. This effort was expanded in much great detail, as shown in her dissertation [Asuncion 2009], and documented on her research seminar held at the Institute for Software Research in June 2009. See http://www.ics.uci.edu/~hasuncio/assets/ISR09_License.pdf.

Our third paper, "Intellectual Property Rights Requirements for Heterogeneously Licensed Systems" in *Proc. 17th. Intern. Conf. Requirements Engineering (RE09)*, Atlanta, GA, 24-33, September 2009, likely represents the major research results to date during the 2009 project period. This study introduces the formal logic scheme we developed to specify heterogeneous OA software licenses in a form that can subsequently be analyzed using the computational tools and techniques developed by Asuncion [2009]. In this regard, this study demonstrates the viability of our approach to be able to verify and validate the rights and obligations associated with different software licenses that are found in complex OA with OSS system components. Furthermore, the study demonstrates that this approach can be applied to OA at different stages in the development and deployment, thus increasing the value of the approach and its potential contribution.

Our fourth and last paper, "The Role of Software Licenses in Open Architecture Ecosystems," *Intern. Workshop on Software Ecosystems*, Intern. Conf. Software Reuse, Falls Church, VA, September 2009, demonstrates how our approach can be further extended to address the analysis of software licenses challenges across the supply chain of commercial and governmental software system contractors and customers. This also increases the value of the approach and its potential contribution.

Final Remarks

Each of the four research papers that constitute the bulk of our project deliverables follow in the remaining parts of this report. We welcome the opportunity to respond to any questions or comments following from our research efforts, along with a willingness to describe our view of how these results can be applied to future open architecture systems that contain software system components that may include those subject to different, possibly conflicting, open source software licenses.

References

- Asuncion, H. (2009). Architecture-Centric Traceability for Stakeholders, unpublished Doctoral Dissertation, Department of Informatics, School of Information and Computer Sciences, University of California, Irvine, December 2009. Related Web site: http://www.isr.uci.edu/~hasuncio/acts/

- Berzins, V. (2008). Which Unchanged Components to Retest after a Technology Upgrade, *Proc. 5th Annual Acquisition Research Symposium*, NPS-AM-08-031, Naval Postgraduate School, Monterey, CA, May.

- Justice, Brig. General Nick (2007). *Deploying Open Technologies and Architectures within Military Systems*, Presentation at 3rd DoD Open Conference, Deployment of Open Technologies and architectures within Military Systems, AFEI Symposium, Arlington VA, 12 December.

- Naegle, B. and Petross, D. (2007). Software Architecture: Managing Design for Achieving Warfighter Capability, *Proc. 5th Annual Acquisition Research Symposium*, NPS-AM-07-104, Naval Postgraduate School, Monterey, CA, May.

- Navy (2007), *Naval Open Architecture Contract Guidebook (V. 1.1),* 6 December 2007.

- Riechers, C., (2007). *The Role of Open Technology in Improving USAF Software Acquisition*, Presentation at "Open Source - Open Standards - Open Architecture**,"** AFEI Symposium, Arlington VA, 14 March 2007.

- Scacchi, W., (2002). Understanding the Requirements for Developing Open Source Software Systems, *IEE Proceedings--Software*, 149(1), 24-39, February 2002.

- Scacchi, W., (2006). Understanding the Evolution of Free/Open Source Software, in N.H. Madhavji, J.F. Ramil and D. Perry (eds.), *Software Evolution and Feedback: Theory and Practice*, 181-206, John Wiley and Sons Inc, New York, 2006.

- Scacchi, W., (2007). Free/Open Source Software Development: Recent Research Results and Methods, in M. Zelkowitz (Ed.), *Advances in Computers*, 69, 243-295, 2007.

- Scacchi, W. and Alspaugh, T., (2008). Emerging Issues in the Acquisition of Open Source Software within the U.S. Department of Defense, *Proc. 5th Annual Acquisition Research Symposium*, NPS-AM-08-036, Naval Postgraduate School, Monterey, CA, May.

- Starrett, E. (2007). Software Acquisition in the Army, *Crosstalk: The Journal of Defense Software Engineering*, 4-8, May, http://stsc.hill.af.mil/crosstalk.

- Weathersby, J.M., (2007). Open Source Software and the Long Road to Sustainability within the U.S. DoD IT System, *The DoD Software Tech News*, 10(2), 20-23, June.

- Wheeler, D.A., (2007). Open Source Software (OSS) in U.S. Government Acquisitions, *The DoD Software Tech News*, 10(2), 7-13, June.

Software Licenses, Open Source Components, and Open Architectures

Thomas A. Alspaugh, Hazeline U. Asuncion, and Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
{alspaugh,hasuncion,wcacchi}@ics.uci.edu

Abstract

A substantial number of enterprises and independent software vendors are adopting a strategy in which software-intensive systems are developed with an open architecture (OA) that may contain open source software (OSS) components or components with open APIs. The emerging challenge is to realize the benefits of openness when components are subject to different copyright or property licenses. In this paper we identify key properties of OSS licenses, present a license analysis scheme to identify license conflicts arising from composed software elements, and apply it to provide guidance for software architectural design choices whose goal is to enable specific licensed component configurations. Our scheme has been implemented in an operational environment and demonstrates a practical, automated solution to the problem of determining overall rights and obligations for alternative OAs.

1. Introduction

It has been common for OSS projects to require that developers contribute their work under conditions that ensure the project can license its products under a specific OSS license. For example, the Apache Contributor License Agreement grants enough rights to the Apache Software Foundation for the foundation to license the resulting systems under the Apache License. This sort of license configuration, in which the rights to a system's components are homogenously granted and the system has a well-defined OSS license, was the norm and continues to this day.

However, we more and more commonly see a different license configuration, in which the components of a system do not have the same license. The resulting system may not have any recognized OSS license at all—in fact, our research indicates this is the most likely outcome—but instead, if all goes well in its design, there will be enough rights available in the system so that it can be used and distributed, and perhaps modified by others and sublicensed, if the corresponding obligations are met. These obligations are likely to differ for components with different licenses; a BSD (Berkeley Software Distribution) licensed component must preserve its copyright notices when made part of the system, for example, while the source code for a modified component covered by MPL (the Mozilla Public License) must be made public, and a component with a reciprocal license such as the Free Software Foundation's GPL (General Public License) might carry the obligation to distribute the source code of that component but also of other components that constitute "a whole which is a work based on" the GPL'd component. 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 license's prohibition of publishing source code, in which case there may be no rights available for the system as a whole, not even the right of use, because the obligations of the licenses that would permit use of its components cannot simultaneously be met.

The central problem we examine and explain in this paper is to identify principles of software architecture and software licenses that facilitate or inhibit success of the OA strategy when OSS and other software components with open APIs are employed. This is the knowledge we seek to develop and deliver. Without such knowledge, it is unlikely that an OA that is clean, robust, transparent, and extensible can be readily produced. On a broader scale, this paper seeks to explore and answer the following kinds of research questions:

- What license applies to an OA system composed with components with different licenses?
- How do alternative OSS licenses facilitate or inhibit the development of OA systems?

- How should software license constraints be specified so it is possible to automatically determine the overall set of rights and obligations associated with a configured software system architecture?

This paper may help establish a foundation for how to analyze and evaluate dependencies that might arise when seeking to develop software systems that embody an OA when different types of software components or software licenses are being considered for integration into an overall system configuration.

In the remainder of this paper, we examine software licensing constraints. This is followed by an analysis of how these constraints can interact in order to determine the overall license constraints applicable to the configured system architecture. Next, we describe an operational environment that demonstrates automatic determination of license constraints associated with a configured system architecture, and thus offers a solution to the problem we face. We close with a discussion of the conclusions that follow.

2. Background

There is little explicit guidance or reliance on systematic empirical studies for how best to develop, deploy, and sustain complex software systems when different OA and OSS objectives are at hand. Instead, we find narratives that provide ample motivation and belief in the promise and potential of OA and OSS without consideration of what challenges may lie ahead in realizing OA and OSS strategies. Ven [2008] is a recent exception.

We believe that a primary challenge to be addressed is how to determine whether a system, composed of subsystems and components each with specific OSS or proprietary licenses, and integrated in the system's planned configuration, is or is not open, and what license constraints apply to the configured system as a whole. This challenge comprises not only evaluating an existing system at run-time, but also at design-time and build-time for a proposed system to ensure that the result is "open" under the desired definition, and that only the acceptable licenses apply; and also understanding which licenses are acceptable in this context. Because there are a range of types and variants of licenses [cf. OSI 2008], each of which may affect a system in different ways, and because there are a number of different kinds of OSS-related components and ways of combining them that affect the licensing issue, a first necessary step is to understand the kinds of software elements that constitute a software architecture, and what kinds of licenses may encumber these elements or their overall configuration.

OA seems to simply mean software system architectures incorporating OSS components and open application program interfaces (APIs). But not all software system architectures incorporating OSS components and open APIs will produce an OA, since the openness of an OA depends on: (a) how/why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, (c) whether the copyright (Intellectual Property) licenses assigned to different OSS components encumber all/part of a software system's architecture into which they are integrated, and (d) the fact that many alternative architectural configurations and APIs exist that may or may not produce an OA [cf. Antón and Alspaugh 2007, Scacchi and Alspaugh 2008]. Subsequently, we believe this can lead to situations in which new software development or acquisition requirements stipulate a software system with an OA and OSS, but the resulting software system may or may not embody an OA. This can occur when the architectural design of a system constrains system requirements—raising the question of what requirements can be satisfied by a given system architecture, when requirements stipulate specific types or instances of OSS (e.g., Web browsers, content management servers) to be employed, or what architecture style [Bass, Clements, and Kazman 2003] is implied by a given set of system requirements.

Thus, given the goal of realizing an OA and OSS strategy together with the use of OSS components and open APIs, it is unclear how to best align acquisition, system requirements, software architectures, and OSS elements across different software license regimes to achieve this goal [Scacchi and Alspaugh 2008].

3. Understanding open architectures

The statement that a system is intended to embody an open architecture using open software technologies like OSS and APIs, does not clearly indicate what possible mix of software elements may be configured into such a system. To help explain this, we first identify what kinds of software elements are included in common software architectures whether they are open or closed [cf. Bass, Clements, Kazman 2003].

1. Software source code components – (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code (e.g., C shell scripts) and (d) intra-application script code (e.g., to create Rich Internet Applications using domain-specific languages (e.g., XUL for Firefox Web

browser [Feldt 2007] or “mashups” [Nelson and Churchill 2006]).
2. Executable components -- These are programs for which the software is in binary form, and its source code may not be open for access, review, modification, and possible redistribution. Executable binaries can be viewed as “derived works” [Rosen 2005].
3. Application program interfaces/APIs – The availability of externally visible and accessible APIs to which independently developed components can be connected is the minimum condition required to form an “open system” [Meyers and Obendorf 2001].
4. Software connectors – In addition to APIs, these may be software either from libraries, frameworks, or application script code whose intended purpose is to provide a standard or reusable way of associating programs, data repositories, or remote services through common interfaces. The High Level Architecture (HLA) is an example of a software connector scheme [Kuhl, Weatherly, Damann 2000], as are CORBA, Microsoft's .NET, Enterprise Java Beans, and LGPL libraries.
5. Configured system or sub-system architectures – These are software systems that can be built to conform to an explicit architectural design. They include software source code components, executable components, APIs, and connectors that are organized in a way that may conform to a known “architectural style” such as the Representational State Transfer [Fielding and Taylor 2002] for Web-based client-server applications, or may represent an original or ad hoc architectural pattern [Bass 2003]. Each of the software elements, and the pattern in which they are arranged and interlinked, can all be specified, analyzed, and documented using an Architecture Description Language and ADL-based support tools [Bass 2003, Medvidovic 1999].

Figure 1 provides an overall view of an archetypal software architecture for a configured system that includes and identifies each of the software elements above, as well as including free/open source software (e.g., Gnome Evolution) and closed source software (WordPerfect) components. In simple terms, the configured system consists of software components (grey boxes in the Figure) that include a Mozilla Web browser, Gnome Evolution email client, and WordPerfect word processor, all running on a Linux operating system that can access file, print, and other remote networked servers (e.g. an Apache Web server). These components are interrelated through a set of software connectors (ellipses in the Figure) that connect the interfaces of software components (small white boxes attached to a component) and link them together. Modern day enterprise systems or command and control systems will generally have more complex architectures and a more diverse mix of software components than shown in the figure here. As we examine next, even this simple architecture raises a number of OSS licensing issues that constrain the extent of openness that may be realized in a configured OA.

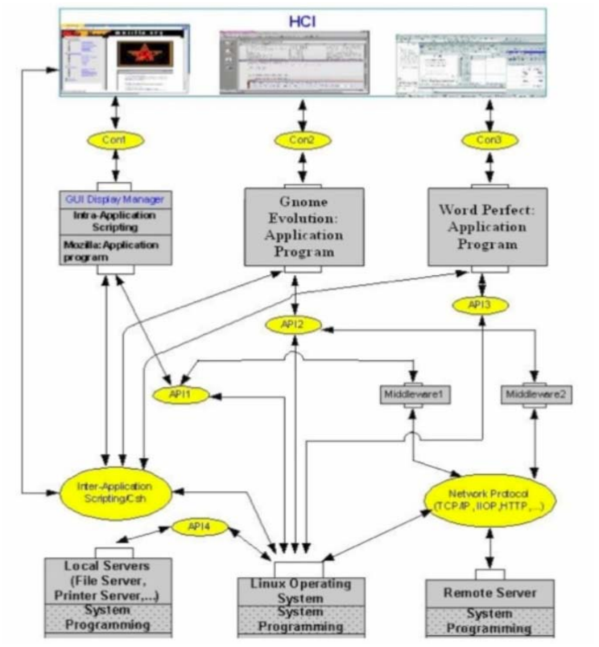

HCI
Con1
Con2
Con3
GUI Display Manager
Intra-Application Scripting
Mozilla: Application program
Gnome Evolution: Application Program
Word Perfect: Application Program
API3
API2
API1
Middleware1
Middleware2
Inter-Application Scripting/Csh
Network Protocol (TCP/IP, IIOP, HTTP,...)
API4
Local Servers (File Server, Printer Server,...)
System Programming
Linux Operating System
System Programming
Remote Server
System Programming

Figure 1. An archetypal software architecture depicting components (grey boxes), connectors (ellipses), interfaces (small boxes on components), and data/control links

4. Understanding open software licenses

A particularly knotty challenge is the problem of licenses in OSS and OA. There are a number of different OSS licenses, and their number continues to grow. Each license stipulates different constraints attached to software components that bear it. External references are available which describe and explain many different licenses that are now in use with OSS [Fontana 2008, OSI 2008, Rosen 2005, St. Laurent 2004].

More and more software systems are designed, built, released, and distributed as OAs composed of components from different sources, some proprietary and others not. Systems include components that are statically bound or interconnected at build-time, while other components may only be dynamically linked for execution at run-time, and thus might not be included as part of a software release or distribution. Software components in such systems evolve not only by ongoing maintenance, but also by architectural refactoring, alternative component interconnections, and component replacement (via maintenance patches, installation of new versions, or migration to new technologies). Software components in such systems may be subject to different software licenses, and later versions of a component may be subject to different licenses (e.g., from CDDL (Sun's Common Development and Distribution License) to GPL, or from GPLv2 to GPLv3).

Software systems with open architectures are subject to different software licenses than may be common with traditional proprietary, closed source systems from a single vendor. Software architects/developers must increasingly attend to how they design, develop, and deploy software systems that may be subject to multiple, possibly conflicting software licenses. We see architects, developers, software acquisition managers, and others concerned with OAs as falling into three groups. The first group pays little or no heed to license conflicts and obligations; they simply focus on the other goals of the system. Those in the second group have assets and resources, and to protect these they may have an army of lawyers to advise them on license issues and other potential vulnerabilities; or they may constrain the design of their systems so that only a small number of software licenses (possibly just one) are involved, excluding components with other licenses independent of whether such components represent a more effective or more efficient solution. The third group falls between these two extremes; members of this group want to design, develop, and distribute the best systems possible, while respecting the constraints associated with different software component licenses. Their goal is a configured OA system that meets all its goals, and for which all the license obligations for the needed copyright rights are satisfied. It is this third group that needs the guidance the present work seeks to provide.

There has been an explosion in the number, type, and variants of software licenses, especially with open source software (cf. OSI 2008). Software components are now available subject to licenses such as the General Public License (GPL), Mozilla Public License (MPL), Apache Public License, (APL), Academic licenses (e.g., BSD, MIT), Creative Commons, Artistic, and others as well as Public Domain (either via explicit declaration or by expiration of prior copyright license). Furthermore, licenses such as these can evolve, resulting in new license versions over time. But no matter their diversity, software licenses represent a legally enforceable contract that is recognized by government agencies, corporate enterprises, individuals, and judicial courts, and thus they cannot be taken trivially. As a consequence, software licenses constrain open architectures, and thus architectural design decisions.

So how might we support the diverse needs of different software developers, with respect to their need to design, develop, and deploy configured software systems with different, possibly conflicting licenses for the software components they employ? Is it possible to provide automated means for helping software developers determine what constraints will result at design-time, build-time, or run-time when their configured system architectures employ diverse licensed components? These are the kind of questions we address in this paper.

4.1. Software licenses: Rights and obligations

Copyright, the common basis for software licenses, gives the original author of a work certain exclusive rights, which for software include the right to use, copy, modify, merge, publication, distribution, sub-licensing, and sell copies. These rights may be licensed to others; the rights may be licensed individually or in groups, and either exclusively so that no one else can exercise them or (more commonly) non-exclusively. After a period of years, the rights enter the public domain, but until then the only way for anyone other than the author to have any of the copyright rights is to license them.

Licenses may impose obligations that must be met in order for the licensee to realize the assigned rights. Commonly cited obligations include the obligation to buy a legal copy to use and not distribute copies (proprietary licenses); the obligation to preserve copyright and license notices (academic licenses); the obligation to publish at no cost source code you modify (MPL); or the reciprocal obligation to publish all source code included at build-time or statically linked (GPL).

Licenses may provide for the creation of derivative works (e.g., a transformation or adaptation of existing software) or collective works (e.g., a Linux distribution that combines software from many independent sources) from the original work, by granting those rights possibly with corresponding obligations.

In addition, the author of an original work can make it available under more than one license, enabling the work's distribution to different audiences with different needs. For example, one licensee might be happy to pay a license fee in order to be able to distribute the work as part of a proprietary product whose source code is not published, while another might need to license the work under MPL rather than GPL in order to have consistent licensing across a system. Thus we now see software distributed under any one of several licenses, with the licensee choosing from two ("dual license") or three (Mozilla's "tri-license") licenses.

The basic relationship between software license rights and obligations can be summarized as follows: if you meet the specified obligations, then you get the specified rights. So, informally, for the academic licenses, if you retain the copyright notice, list of license conditions, and disclaimer, then you can use, modify, merge, sub-license, etc. For MPL, if you publish modified source code and sub-licensed derived works under MPL, then you get all the MPL rights. And so forth for other licenses. However, one thing we have learned from our efforts to carefully analyze and lay out the obligations and rights pertaining to each license is that license details are difficult to comprehend and track—it is easy to get confused or make mistakes. Some of the OSS licenses were written by developers, and often these turn out to be incomplete and legally ambiguous; others, usually more recent, were written by lawyers, and are more exact and complete but can be difficult for non-lawyers to grasp. The challenge is multiplied when dealing with configured system architectures that compose multiple components with heterogeneous licenses, so that the need for legal interpretations begins to seem inevitable [cf. Fontana 2008, Rosen 2005]. Therefore, one of our goals is to make it possible to architect software systems of heterogeneously-licensed components without necessarily consulting legal counsel. Similarly, such a goal is best realized with automated support that can help architects understand design choices across components with different licenses, and that can provide support for testing build-time releases and run-time distributions to make sure they achieve the specified rights by satisfying the corresponding obligations.

4.2. Expressing software licenses

Historically, most software systems, including OSS systems, were entirely under a single software license. However, we now see more and more software systems being proposed, built, or distributed with components that are under various licenses. Such systems may no longer be covered by a single license, unless such a licensing constraint is stipulated at design-time, and enforced at build-time and run-time. But when components with different licenses are to be included at build-time, their respective licenses might either be consistent or conflict. Further, if designed systems include components with conflicting licenses, then one or more of the conflicting components must be excluded in the build-time release or must be abstracted behind an open API or middleware, with users required to download and install to enable the intended operation. (This is common in Linux distributions subject to GPL, where for example users may choose to acquire and install proprietary run-time components, like proprietary media players). So a component license conflict need not be a show-stopper if identified at design time. However, developers have to be able to determine which components' licenses conflict and to take appropriate steps at design, build, and run times, consistent with the different concerns and requirements that apply at each phase [cf. Scacchi and Alspaugh 2008].

In order to fulfill our goals, we need a scheme for expressing software licenses that is more formal and less ambiguous than natural language, and that allows us to identify conflicts arising from the various rights and obligations pertaining to two or more component's licenses. We considered relatively complex structures (such as Hohfeld's eight fundamental jural relations [Hohfeld 1913]) but, applying Occam's razor, selected a simpler structure. 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", or "must not", with "may" expressing a right and "must" and "must not" expressing obligations; following Hohfeld, the lack of a right (which would be "may not") correlates with a duty to not exercise the right ("must not"), and whenever lack of a right seemed significant in a license we expressed it as a negative obligation with "must not". The *action* is a verb or verb phrase describing what may, must, or must not be done, with the *object* completing the description. We specify an object separately from the action in order to minimize the set of actions. A license then may be expressed as a set of rights, with each right associated (in that license) with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 2 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.

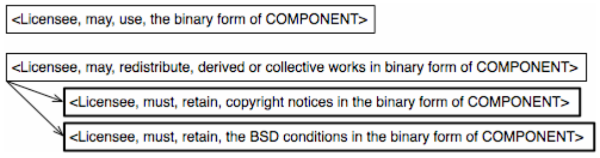


Figure 2. A portion of the BSD license tuples

We now turn to examine how OA software systems that include components with different licenses can be designed and analyzed while effectively tracking their rights and obligations.

When designing an OA software system, there are heuristics that can be employed to enable architectural design choices that might otherwise be excluded due to license conflicts. First, it is possible to employ a "license firewall" which serves to limit the scope of reciprocal obligations. Rather than simply interconnecting conflicting components through static linking of components at build time, such components can be logically connected via dynamic links, client-server protocols, license shims (e.g., via LGPL connectors), or run-time plug-ins. Second, the source code of statically linked OSS components must be made public. Third, it is necessary to include appropriate notices and publish required sources when academic licenses are employed. 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 become too cumbersome. Thus, automated support needs to be provided to help overcome and manage the multi-component, multi-license complexity.

5. Automating analysis of software license rights and obligation

We find that if we start from a formal specification of a software system's architecture, then we can associate software license attributes with the system's components, connectors, and sub-system architectures and calculate the copyright rights and obligations for the system. Accordingly, we employ an architectural description language specified in xADL [2005] to describe OAs that can be designed and analyzed with a software architecture design environment [Medvidovic 1999], such as ArchStudio4 [2006]. We have taken this environment and extended it with a Software Architecture License Traceability Analysis module [cf. Asuncion 2008]. This allows for the specification of licenses as a list of attributes (license tuples) using a form-based user interface, similar to those already used and known for ArchStudio4 and xADL [ArchStudio 2006, Medvidovic 1999].

Figure 3 shows a screenshot of an ArchStudio4 session in which we have modeled the OA seen in Figure 1. OA software components, each of which has an associated license, are indicated by darker shaded boxes. Light shaded boxes indicate connectors. Architectural connectors may or may not have associated license information; those with licenses (such as architectural connectors that represent functional code) are treated as components during license traceability analysis. A directed line segment indicates a link. Links connect interfaces between the components and connectors. Furthermore, the Mozilla component as shown here contains a hypothetical subarchitecture for modeling the role of intra-application scripting, as might be useful in specifying license constraints for Rich Internet Applications. This subarchitecture is specified in the same manner as the overall system architecture, and is visible in Figure 5. The automated environment allows for tracing and analysis of license attributes and conflicts.

Figure 4 shows a view of the internal XML representation of a software license. Analysis and calculations of rights, obligations, and conflicts for the OA are done in this form. This schematic representation is similar in spirit to that used for specifying and analyzing privacy and security regulations associated with certain software systems [Breaux and Anton 2008].

With this basis to build on, it is now possible to analyze the alignment of rights and obligations for the overall system:

1. Propagation of reciprocal obligations

   Reciprocal obligations are imposed by the license of a GPL'd component on any other component that is part of the same "work based on the Program" (i.e. on the first component), as defined in GPL. We follow the widely-accepted interpretation that build-time static linkage propagate the reciprocal obligations, but the "license firewalls" do not. Analysis begins, therefore, by propagating these obligations along all connectors that are not license firewalls.

2. Obligation conflicts

   An obligation can conflict with another obligation contrary to it, or with the set of available rights, by requiring a copyright right that has not been granted. For instance, the Corel proprietary license for the WordPerfect component, CTL (Corel Transactional License), may be taken to entail that a licensee must not redistribute source code. However, an OSS license, GPL, may state that a licensee must redistribute source code. Thus, the conflict appears in the modality of the two otherwise identical obligations, "must not" in CTL and "must" in GPL. A conflict on the same point could occur also between GPL and a component whose license fails to grant the right to distribute its source code.

   This phase of the analysis is affected by the overall set of rights that are required. If conflicts arise involving the union of all obligations in all components' licenses, it may be possible to eliminate some conflicts by selecting a smaller set of rights, in which case only the obligations for those rights need be considered.

Figure 5 shows a screenshot in which the License Traceability Analysis module has identified obligation conflicts between the licenses of two pairs of components ("WordPerfect" and "Linux OS", and "GUIDisplayManager" and "GUIScript-Interpreter").

3. Rights and obligations calculations

The rights available for the entire system (use, copy, modify, etc.) then are calculated as the intersection of the sets of rights available for each component of the system.

The obligations required for the whole system then are the union of the specific obligations for each component that are associated with those rights. Examples of specific obligations are "Licensee must retain copyright notices in the binary form of `module.c`" or "Licensee must publish the source code of `component.java` version `1.2.3`."

Figure 6 shows a report of the calculations for the hypothetical subarchitecture of the Mozilla component in our archetypal architecture, exhibiting an obligation conflict and the single copyright right (to run the system) that the prototype tool shows would be available for the subarchitecture as a whole if the conflict is resolved; a production tool would also list the rights (none) currently available.

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, employing one or more connectors along the paths between the components that act as a license firewall, thereby mitigating or neutralizing the component-component license conflict. This means that the architecture and the 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.

At build-time (and later at run-time), many of the obligations can be tested and verified, for example that the binaries contain the appropriate notices for their licenses, and that the source files are present in the correct version on the Web. These tests can be generated from the internal list of obligations and run automatically. If the system's interface were extended to add a control for it, the tests could be run by a deployed system.

The prototype License Traceability Analysis module provides a proof-of-concept for this approach. We encoded the core provisions of four licenses in XML for the tool—GPL, MPL, CTL, and AFL (Academic Free License)—to examine the effectiveness of the license tuple encoding and the calculations based upon it. While it is clear that we could use a more complex and expressive structure for encoding licenses, in encoding the license provisions to date we found that the tuple representation was more expressive than needed; for example, the actor was always "licensee" and seems likely to remain so, and we found use for only three operations or modalities. At this writing, the module shows proof of concept for calculating with reciprocal obligations by propagating them to adjacent statically-linked modules; the extension to all paths not blocked by license firewalls is straightforward and is independent of the scheme and calculations described here. Reciprocal obligations are identified in the tool by lookup in a table, and the meaning and scope of reciprocality is hard-coded; this is not ideal, but we considered it acceptable since the legal definition in terms of the reciprocal licenses will not change frequently. We also focused on the design-time analysis and calculation rather than build- or run-time as it involves the widest range of issues, including representations, rights and obligations calculations, and design guidance derived from them.

Based on our analysis approach, it appears that the questions of what license (if any) covers a specific configured system, and what rights are available for the overall system (and what obligations are needed for them) are difficult to answer without automated license-architecture analysis. This is especially true if the system or sub-system is already in operational run-time form [cf. Kazman and Carrière 1999]. It might make distribution of a composite OA system somewhat problematic if people cannot understand what rights or obligations are associated with it. We offer the following considerations to help make this clear. For example, a Mozilla/Firefox Web browser covered by the MPL (or GPL or LGPL, in accordance with the Mozilla Tri-License) may download and run intra-application script code that is covered by a different license. If this script code is only invoked via dynamic run-time linkage, or via a client-server transaction protocol, then there is no propagation of license rights or obligations. However, if the script code is integrated into the source code of the Web browser as persistent part of an application (e.g., as a plug-in), then it could be viewed as a configured sub-system that may need to be accessed for license transfer or conflict implications. Another different kind of example can be anticipated with application programs (like Web browsers, email clients, and word processors) that employ Rich Internet Applications or mashups entailing the use of content (e.g., textual character fonts or geographic maps) that is subject to copyright protection, if the content is embedded in and bundled with the scripted application sub-system. In such a case, the licenses involved may not be limited to OSS or proprietary software licenses.

In the end, it becomes clear that it is possible to automatically determine what rights or obligations are associated with a given system architecture at design-time, and whether it contains any license conflicts that might prevent proper access or use at build-time or run-time, given an approach such as ours.

6. Discussion

Software system configurations in OAs are intended to be adapted to incorporate new innovative software technologies that are not yet available. These system configurations will evolve and be refactored over time at ever increasing rates [Scacchi 2007], components will be patched and upgraded (perhaps with new license constraints), and inter-component connections will be rewired or remediated with new connector types. As such, sustaining the openness of a configured software system will become part of ongoing system support, analysis, and validation. This in turn may require ADLs to include OSS licensing properties on components, connectors, and overall system configuration, as well as in appropriate analysis tools [cf. Bass, Clements, and Kazman 2003, Medvidovic 1999].

Constructing these descriptions is an incremental addition to the development of the architectural design, or alternative architectural designs. But it is still time-consuming, and may present a somewhat daunting challenge for large pre-existing systems that were not originally modeled in our environment.

Advances in the identification and extraction of configured software elements at build time, and their restructuring into architectural descriptions is becoming an ever more automatable endeavor [cf. Choi 1990, Kazman 1999, Jansen 2008]. Further advances in such efforts have the potential to automatically produce architectural descriptions that can either be manually or semi-automatically annotated with their license constraints, and thus enable automated construction and assessment of build-time software system architectures.

The list of recognized OSS licenses is long and ever-growing, and as existing licenses are tested in the courts we can expect their interpretations to be clarified and perhaps altered; the GPL definition of “work based on the Program”, for example, may eventually be clarified in this way, possibly refining the scope of reciprocal obligations. Our expressions of license rights and obligations are for the most part compared for identical actors, actions, and objects, then by looking for “must not” in one and either “must” or “may” in the other, so that new licenses may be added by keeping equivalent rights or obligations expressed equivalently. Reciprocal obligations, however, are handled specially by hard-coded algorithms to traverse the scope of that obligation, so that addition of obligations with different scope, or the revision of the understanding of the scope of an existing obligation, requires development work. Possibly these issues will be clarified as we add more licenses to the tool and experiment with their application in OA contexts.

Lastly, our scheme for specifying software licenses offers the potential for the creation of shared repositories where these licenses can be accessed, studied, compared, modified, and redistributed.

7. Conclusion

The relationship between open architecture, open source software, and multiple software licenses is poorly understood. OSS is often viewed as primarily a source for low-cost/free software systems or software components. Thus, given the goal of realizing an OA strategy together with the use of OSS components and open APIs, it has been unclear how to best align software architecture, OSS, and software license regimes to achieve this goal. Subsequently, the central problem we examined in this paper was to identify principles of software architecture and software copyright licenses that facilitate or inhibit how best to insure the success of an OA strategy when OSS and open APIs are required or otherwise employed. In turn, we presented an analysis scheme and operational environment that demonstrates that an automated solution to this problem exists.

We have developed and demonstrated an operational environment that can automatically determine the overall license rights, obligations, and constraints associated with a configured system architecture whose components may have different software licenses. Such an environment requires the annotation of the participating software elements with their corresponding licenses. These annotated software architectural descriptions can be prescriptively analyzed at design-time as we have shown, or descriptively analyzed at build-time or run-time. Such a solution offers the potential for practical support in design-, build-, and run-time license conformance checking and the ever-more complex problem of developing large software systems from configurations of software elements that can evolve over time.

8. Acknowledgments

The research described in this report has been supported by grants #0534771 from the U.S. National Science Foundation, and Acquisition Research Program at the Naval Postgraduate School. No endorsement implied.

References

Alspaugh, T.A and Antón, A.I., (2007). Scenario Support for Effective Requirements, Information and Software Technology, 50(3), 198-220.

ArchStudio (2006). ArchStudio 4 Software and Systems Architecture Development Environment. Institute for Software Research, University of California, Irvine.
http://www.isr.uci.edu/projects/archstudio/

Asuncion, H. (2008). Towards Practical Software Traceability, in Companion of the 30th Intern. Conf. Software Engineering, 1023-1026, Leipzig, Germany.

Bass, L., Clements, P., and Kazman, R., (2003). Software Architecture in Practice, 2nd Edition, Addison-Wesley Professional, New York..

Breaux, T.D. and Anton, A.I. (2008). Analyzing Regulatory Rules for Privacy and Security Requirements, IEEE Trans. Software Engineering, 34(1), 5-20.

Choi, S. and Scacchi, W. (1990). Extracting and Restructuring the Design of Large Systems, IEEE Software, 7(1), 66-71.

Feldt, K., (2007). Programming Firefox: Building Rich Internet Applications with XUL, O'Reilly Press, Sebastopol, CA.

Fontana, R., Kuhn, B.M., Molgen, E., et al. (2008). A Legal Issues Primer for Open Source and Free Software Projects, Software Freedom Law Center, Version 1.5.1,
http://www.softwarefreedom.org/resources/2008/foss-primer.pdf

Fielding, R. and Taylor, R.N., (2002). Principled Design of the Modern Web Architecture, ACM Transactions Internet Technology, 2(2), 115-150.

Hohfeld, W.N. (1913). Some Fundamental Legal Conceptions as Applied in Judicial Reasoning. Yale Law Journal, 23(1), 16-59.

Jansen, A., Bosch, J., and Avgeriou, P. (2008). Documenting After the Fact: Recovering Architectural Design Decisions, J. Systems and Software, 81(4), 536-557.

Kazman, R. and Carrière, J. (1999). Playing Detective: Reconstructing Software Architecture from Available Evidence. J. Automated Software Engineering, 6(2), 107-138.

Kuhl, F., Weatherly, R., and Dahmann, J., (2000). Creating Computer Simulation Systems: An Introduction to the High Level Architecture, Prentice-Hall PTR, Upper Saddle River, New Jersey.

Medvidovic, N.,Rosenblum, D.S., and Taylor, R.N. (1999). A Language and Environment for Architecture-Based Software Development and Evolution. In Proc. 21st Intern. Conf. Software Engineering (ICSE '99). 44-53, IEEE Computer Society. Los Angeles, CA.

Meyers, B.C. and Obendorf, P., (2001). Managing Software Acquisition: Open Systems and COTS Products, Addison-Wesley, New York.

Nelson L. and Churchill, E.F., (2006). Repurposing: Techniques for Reuse and Integration of Interactive Services, Proc. 2006 IEEE Intern. Conf. Information Reuse and Integration, September.

OSI (2008). The Open Source Initiative,
http://www.opensource.org/

Rosen, L. (2005). Open Source Licensing: Software Freedom and Intellectual Property Law, Prentice-Hall PTR, Upper Saddle River, New Jersey.
http://www.rosenlaw.com/oslbook.htm

Scacchi, W., (2002). Understanding the Requirements for Developing Open Source Software Systems, IEE Proceedings--Software, 149(1), 24-39, February.

Scacchi, W. (2007). Free/Open Source Software Development: Recent Research Results and Emerging Opportunities, Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 459-468.

Scacchi, W. and Alspaugh, T.A. (2008). Emerging Issues in the Acquisition of Open Source Software within the U.S. Department of Defense, Proc. 5th Annual Acquisition Research Symposium, Vol. 1, 230-244, NPS-AM-08-036, Naval Postgraduate School, Monterey, CA

St. Laurent, A.M., (2004). Understanding Open Source and Free Software Licensing, O'Reilly Press, Sebastopol, CA.

Ven, K. and Mannaert, H., (2008). Challenges and Strategies in the Use of Open Source Software by Independent Software Vendors, Information and Software Technology, 50, 991-1002.

Wheeler, D.A., (2007). Open Source Software (OSS) in U.S. Government Acquisitions, The DoD Software Tech News, 10(2), 7-13, June.

xADL (2005). xADL 2.0: Highly-extensible architecture description language for software and systems. Institute for Software Research, University of California, Irvine. http://www.isr.uci.edu/projects/xarchuci/

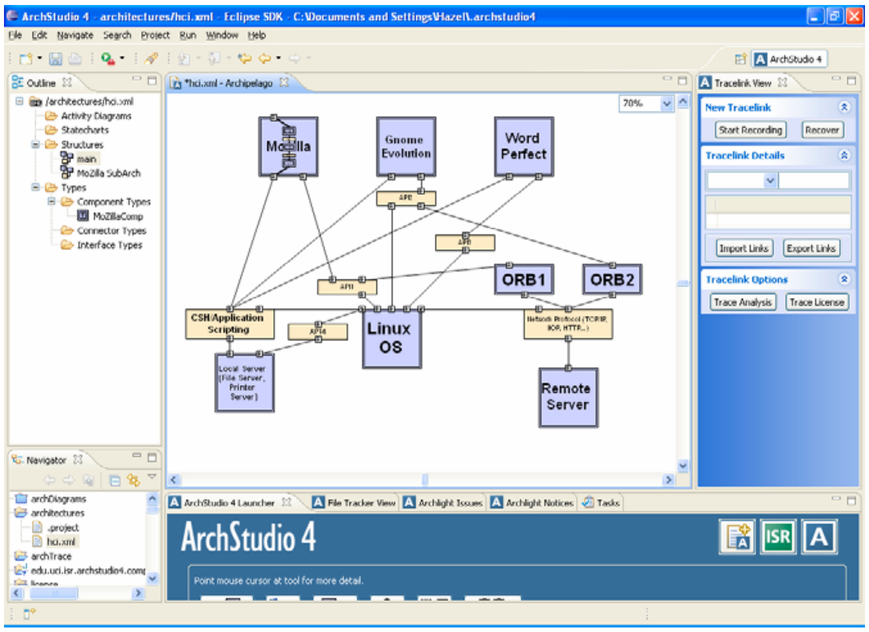
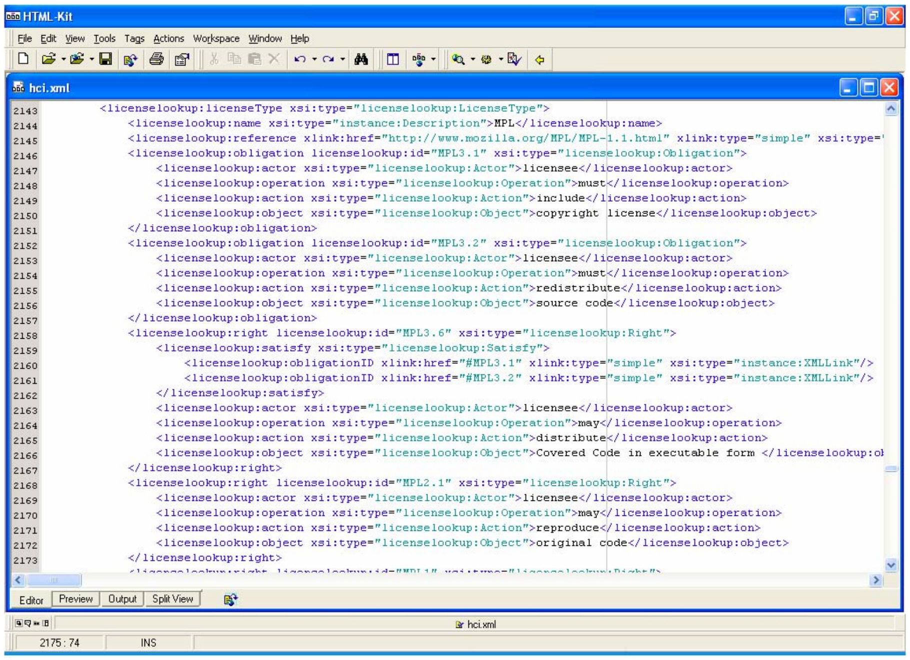
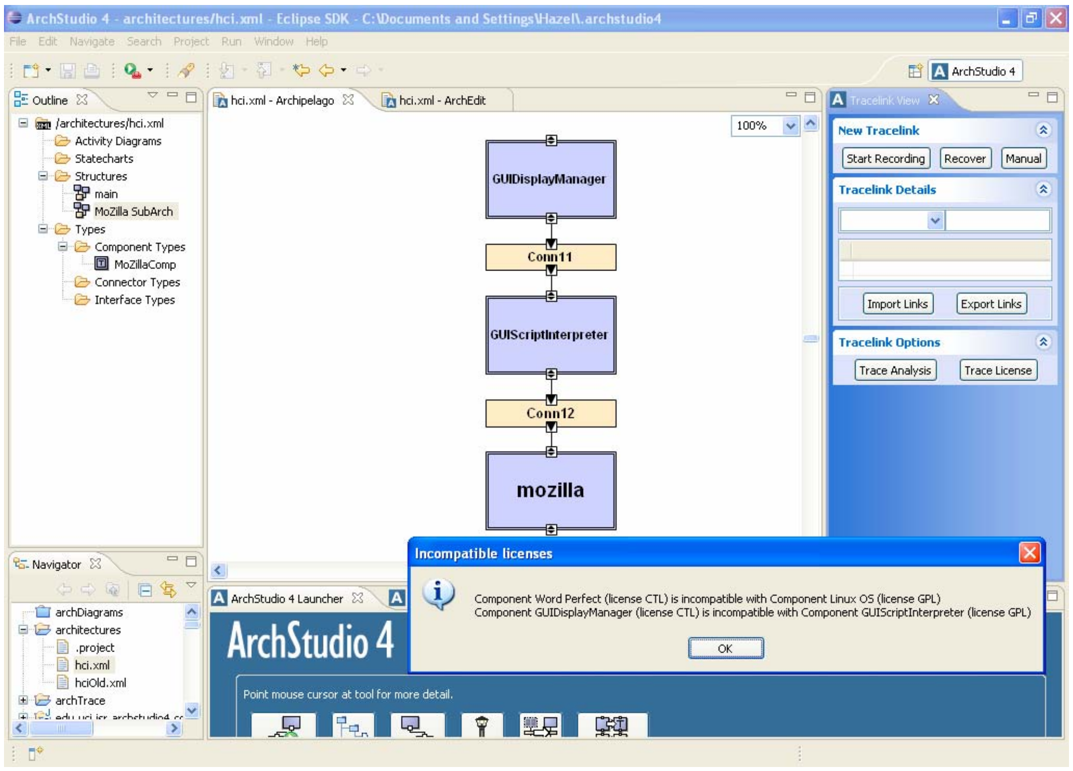


Figure 3. An ArchStudio 4 model of the open software architecture of Figure 1



```
        <licenselookup:licenseType xsi:type="licenselookup:LicenseType">
            <licenselookup:name xsi:type="instance:Description">MPL</licenselookup:name>
            <licenselookup:reference xlink:href="http://www.mozilla.org/MPL/MPL-1.1.html" xlink:type="simple" xsi:type='
            <licenselookup:obligation licenselookup:id="MPL3.1" xsi:type="licenselookup:Obligation">
                <licenselookup:actor xsi:type="licenselookup:Actor">licensee</licenselookup:actor>
                <licenselookup:operation xsi:type="licenselookup:Operation">must</licenselookup:operation>
                <licenselookup:action xsi:type="licenselookup:Action">include</licenselookup:action>
                <licenselookup:object xsi:type="licenselookup:Object">copyright license</licenselookup:object>
            </licenselookup:obligation>
            <licenselookup:obligation licenselookup:id="MPL3.2" xsi:type="licenselookup:Obligation">
                <licenselookup:actor xsi:type="licenselookup:Actor">licensee</licenselookup:actor>
                <licenselookup:operation xsi:type="licenselookup:Operation">must</licenselookup:operation>
                <licenselookup:action xsi:type="licenselookup:Action">redistribute</licenselookup:action>
                <licenselookup:object xsi:type="licenselookup:Object">source code</licenselookup:object>
            </licenselookup:obligation>
            <licenselookup:right licenselookup:id="MPL3.6" xsi:type="licenselookup:Right">
                <licenselookup:satisfy xsi:type="licenselookup:Satisfy">
                    <licenselookup:obligationID xlink:href="#MPL3.1" xlink:type="simple" xsi:type="instance:XMLLink"/>
                    <licenselookup:obligationID xlink:href="#MPL3.2" xlink:type="simple" xsi:type="instance:XMLLink"/>
                </licenselookup:satisfy>
                <licenselookup:actor xsi:type="licenselookup:Actor">licensee</licenselookup:actor>
                <licenselookup:operation xsi:type="licenselookup:Operation">may</licenselookup:operation>
                <licenselookup:action xsi:type="licenselookup:Action">distribute</licenselookup:action>
                <licenselookup:object xsi:type="licenselookup:Object">Covered Code in executable form </licenselookup:ol
            </licenselookup:right>
            <licenselookup:right licenselookup:id="MPL2.1" xsi:type="licenselookup:Right">
                <licenselookup:actor xsi:type="licenselookup:Actor">licensee</licenselookup:actor>
                <licenselookup:operation xsi:type="licenselookup:Operation">may</licenselookup:operation>
                <licenselookup:action xsi:type="licenselookup:Action">reproduce</licenselookup:action>
                <licenselookup:object xsi:type="licenselookup:Object">original code</licenselookup:object>
            </licenselookup:right>
```

Figure 4. A view of the internal schematic representation of the Mozila Public License

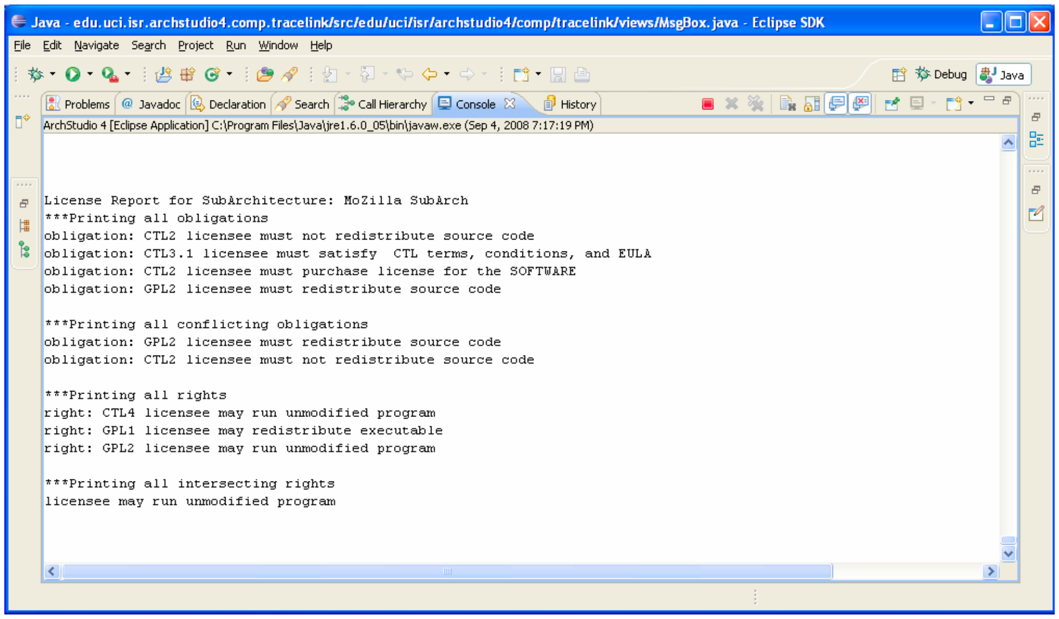


Incompatible licenses

Component Word Perfect (license CTL) is incompatible with Component Linux OS (license GPL)
Component GUIDisplayManager (license CTL) is incompatible with Component GUIScriptInterpreter (license GPL)

OK

Figure 5. License conflicts have been identified between two pairs of components


Java - edu.uci.isr.archstudio4.comp.tracelink/src/edu/uci/isr/archstudio4/comp/tracelink/views/MsgBox.java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Debug
Java
Problems
Javadoc
Declaration
Search
Call Hierarchy
Console
History

ArchStudio 4 [Eclipse Application] C:\Program Files\Java\jre1.6.0_05\bin\javaw.exe (Sep 4, 2008 7:17:19 PM)

```
License Report for SubArchitecture: MoZilla SubArch
***Printing all obligations
obligation: CTL2 licensee must not redistribute source code
obligation: CTL3.1 licensee must satisfy  CTL terms, conditions, and EULA
obligation: CTL2 licensee must purchase license for the SOFTWARE
obligation: GPL2 licensee must redistribute source code

***Printing all conflicting obligations
obligation: GPL2 licensee must redistribute source code
obligation: CTL2 licensee must not redistribute source code

***Printing all rights
right: CTL4 licensee may run unmodified program
right: GPL1 licensee may redistribute executable
right: GPL2 licensee may run unmodified program

***Printing all intersecting rights
licensee may run unmodified program
```

Figure 6. A report identifying the obligations, conflicts, and rights for the architectural model

Analyzing Software Licenses in Open Architecture Software Systems

Thomas A. Alspaugh
Department of Computer Science
Georgetown University
Washington, DC 20057 USA
alspaugh@cs.georgetown.edu

Hazeline U. Asuncion and Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
{hasuncion,wscacchi}@ics.uci.edu

Abstract

A substantial number of enterprises and independent software vendors are adopting a strategy in which software-intensive systems are developed with an open architecture (OA) that may contain open source software (OSS) components or components with open APIs. The emerging challenge is to realize the benefits of openness when components are subject to different copyright or property licenses. In this position paper, we identify key properties of OSS licenses, present a license analysis scheme, and discuss our approach for automatically analyzing license interactions.

1. Introduction

Open architectures have generally referred to the ability to use third party components to create a software system. Oreizy uses the term to refer to his customization technique of making the architecture model an explicit and malleable part of the deployed system [16], while the Department of Defense community uses the term to refer to guidelines on acquiring and composing third party components into a software system [18]. Today, we see more and more software-intensive systems developed using an OA strategy not only with open source software (OSS) components but also proprietary components with open APIs (e.g. [20]). Developing systems using the OA technique can lower development costs [18]. Composing a system with heterogeneously-licensed components, however, increases the likelihood of liabilities stemming from incompatible licenses. Thus, in this paper, 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.*

OA systems were formerly composed solely of homogenously-licensed OSS components. These OSS projects have commonly required developers to contribute their work under conditions that ensure the project can license its products under a specific OSS license. This is changing, however. More systems are being composed of software components associated with different licenses. The resulting system may not have any recognized OSS license at all—but if the system is designed well and if the corresponding obligations are met, copyright rights may be available to allow its redistribution and sublicensing.

Due to the sheer number of license types, variants, versions, and the various stipulations attached to each of these licenses, analyzing the compatibility or lack thereof between the various licenses in a system is extremely difficult. Licenses are often incomplete or hard to understand. Licenses are also legally binding.

Thus, we aim to identify principles of software architecture and software licenses that facilitate success of an OA system. We present a systematic approach to analyzing license interaction within a system using a formal license model that can adequately express the majority of current license types. We then incorporate this model into xADL, an extensible architecture description language that rigorously represents a software system [10]. We discuss our automated support for analyzing licenses within ArchStudio4 [11].

2. Background

There is little explicit guidance on how best to develop, deploy, and sustain complex software systems when different OA and OSS objectives are at hand. Ven [21] and German [8] are recent exceptions.

OA may simply seem to mean software system architectures incorporating OSS components and open application program interfaces (APIs). But not all software system architectures incorporating OSS components and open APIs will produce an OA, since the available license rights of an OA depend on: (a) how/why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, (c) whether the licenses of different OSS components encumber all/part of a software system's architecture into which they are integrated, and (d) the fact that many alternative architectural configurations and APIs exist that may or may not produce an OA system (cf. [3, 18]). Thus, new software development or acquisition requirements may stipulate a software system with an OA and OSS, but the resulting system may or may not have the rights needed to embody an OA system.

3. Understanding open architectures

Stating that an OA system comprises OSS and open API components does not clearly indicate what possible mixes of software elements may be configured into such a system. To help explain this, we first identify software elements included in common software architectures that affect whether they are open or closed [5].

Software source code components – These can be either (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code (e.g., C shell scripts) and (d) intra-application script code (e.g., to create Rich Internet Applications using domain-specific languages such as XUL for Firefox Web browser [6] or "mashups" [15]).

Executable components -- These are programs in binary form, and its source code may not be open for access, review, modification, and possible redistribution. Executable binaries are a compilation of source code and they can be viewed as "derived works" [17].

Application program interfaces/APIs – The availability of externally visible and accessible APIs is the minimum requirement to form an "open system" [14].

Software connectors – Software intended to provide a standard or reusable way of communication through common interfaces, e.g. High Level Architecture (HLA) [12], CORBA, MS .NET, and GNU Lesser General Public License (LGPL) libraries.

Configured system or sub-system architectures – These are software systems which may comprise of components with different licenses, affecting the overall system license. To minimize license interaction, a configured system or sub-architecture may be surrounded by a *license firewall,* a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of reciprocal obligations. The Affero General Public License (AGPL) [2] prohibits using license firewalls.

4. Understanding open software licenses

A particularly knotty challenge is the problem of heterogeneous licenses in software systems. There has been an explosion in the number, type, and variants of software licenses, especially with open source software (cf. [1]). License types include General Public License (GPL), Mozilla Public License (MPL), Apache Public License, (APL), academic licenses such as Berkeley Software Distribution (BSD) and MIT, Creative Commons, Artistic, and Public Domain (either via explicit declaration or by expiration of prior copyright license). Within each license types are numerous variants. Furthermore, licenses can evolve, resulting in new license versions over time. Finally, each license stipulates different constraints to software components that bear it. Discussions of many different licenses currently used with OSS are available [1, 7, 17, 19].

The way components are configured also affects the license of the overall system. Furthermore, the component configurations at build-time and run-time may have different license implications. For instance, components may be statically bound or interconnected at build-time, while other components may only be dynamically linked for execution at run-time, and thus might not be included as part of a software release or distribution. On top of this, software maintenance such as architectural refactoring, alternative component interconnections, and component replacement (via maintenance patches, installation of new versions, or migration to new technologies) can all have effects on the overall license of the system.

4.1. Software licenses: rights and obligations

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

Licenses may impose obligations that must be met in order for the licensee to realize the assigned rights. Commonly cited obligations include the obligation to publish at no cost the 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 license's prohibition of publishing source code. In this case, rights may not be available for the system as a whole, not even the right of use, because the two obligations cannot simultaneously be met.

The basic relationship between software license rights and obligations can be summarized as follows: if the specified obligations are met, then the specified rights are granted. For example, if you publish modified source code and sub-licensed derived works under MPL, then you get all the MPL rights for the original and modified code. However, license details are difficult to comprehend and track—it is easy to get confused or make mistakes. Licenses written by developers are often incomplete and legally ambiguous, while those written by lawyers, are more exact and complete but can be difficult for non-lawyers to grasp. The challenge is multiplied when dealing with configured systems that compose multiple components with heterogeneous licenses, so that the need for legal interpretations begins to seem inevitable (cf. [7, 17]).

4.2. Expressing software licenses

We propose a scheme for expressing software licenses that is more formal and less ambiguous than natural language, and that allows us to identify conflicts arising from the various rights and obligations pertaining to two or more component's licenses. We considered relatively complex structures (such as Hohfeld's eight fundamental jural relations [9]) but, applying Occam's razor, selected a simpler structure. 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", or "must not", with "may" expressing a right and "must" and "must not" expressing obligations. A copyright right is only available to entities who have been granted a sublicense. Thus, 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, or must not be done, with the *object* completing the description. We specify an object separately from the action to minimize the set of actions. 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 1 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.

When designing an OA software system, there are heuristics that can be employed to enable architectural design choices that might otherwise be excluded due to license conflicts. First, it is possible to employ a license firewall that serves to limit the scope of reciprocal obligations. Rather than simply interconnecting conflicting components through static linking of components at build-time, such components can be logically connected via dynamic links, client-server protocols, license shims (e.g., via LGPL connectors), or run-time plug-ins. Second, the source code of statically linked OSS components must be made public. Third, it is necessary to include appropriate notices and publish required sources when academic licenses are employed. However, even using design heuristics such as these (and there are many), keeping track of license rights and obligations across interconnected components in complex OAs quickly become too cumbersome. Thus, automated support is needed to manage the multi-component, multi-license complexity.

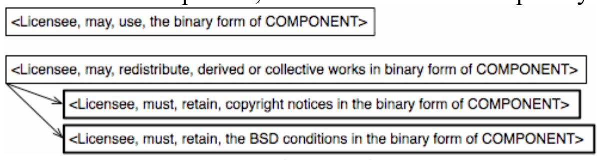


Figure 1. A portion of the BSD license tuples

5. Automating software license analysis

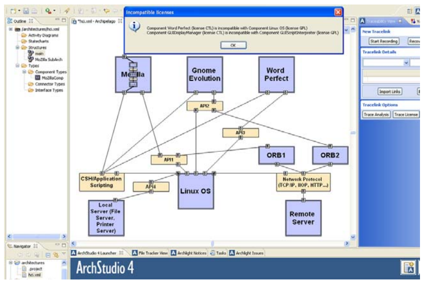
If we start from a formal specification of a software system's architecture, we can associate software license attributes with the system's components, connectors, and sub-system architectures and calculate the copyright rights and obligations for the system's configuration. Accordingly, we use an architectural description language specified in xADL [10] to describe OAs that can be designed and analyzed with a software architecture design environment [13], such as ArchStudio4 [11]. ArchStudio4 currently has software traceability tool support (cf. [4]) and we have extended it with a Software Architecture License Traceability Analysis module (see Fig 2). This allows for the specification of licenses as a list of attributes (license tuples) using a form-based user interface in ArchStudio4.

We analyze rights and obligations as follows:

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

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 a proprietary software and "must" in GPL.

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 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.


Incompatible licenses

Component Word Perfect (license CTL) is incompatible with Component Linux OS (license GPL)
Component GUIDisplayManager (license CTL) is incompatible with Component GUIScriptInterpreter (license GPL)

Figure 2: License traceability analysis tool

6. Ongoing work

We are currently encoding major license types such as GPL, MPL, CTL to examine the effectiveness of the license tuple encoding and the calculations based upon it. Thus far, we are finding that the tuple representation is sufficiently expressive for our needs. We are also currently evaluating the effectiveness of our automated license analysis on an actual heterogeneously licensed system. In addition, we are exploring the impact of patent and other provisions in licenses. Finally, we are studying how the design time and build-time analysis of component configuration relates to the eventual run-time license of a system.

7. Acknowledgments

Effort funded by grants 0534771 and 0808783 from the U.S. NSF & Acquisition Research Program at the Naval Postgraduate School. No endorsement implied.

8. References

[1] *Open Source Initiative*. http://www.opensource.org, 2008.

[2] Affero Inc. *Affero General Public License*. http://www.affero.org/oagpl.html, 2007.

[3] Alspaugh, T.A. and Antón, A.I. Scenario Support for Effective Requirements. *Information and Software Technology*. 50(3), p. 198-220, February, 2007.

[4] Asuncion, H. Towards Practical Software Traceability. In *Proc. of the 30th International Conf on Software Engineering Doctoral Symposium*. Leipzig, Germany, 2008.

[5] Bass, L., Clements, P., et al. *Software Architecture in Practice*. 2nd ed. Addison-Wesley Prof: New York, 2003.

[6] Feldt, K. *Programming Firefox: Building Rich Internet Applications with XUL*.O'Reilly Press: Sebastopol, CA, 2007.

[7] Fontana, R., Kuhn, B.M., et al. *A Legal Issues Primer for Open Source and Free Software Projects*. http://www.sofwarefreedom.org/resources/2008/foss-primer.pdf, Software Freedom Law Center, Report Version 1.5.1, 2008.

[8] German, D.M. and Hassan, A.E. License Integration Patterns: Dealing with Licenses Mismatches in Component-Based Development. In *Proc. of the 31st International Conference on Software Engineering (ICSE 2009)*. Vancouver, Canada, May 16-24, 2009.

[9] Hohfeld, W.N. Some Fundamental Legal Conceptions as Applied in Judicial Reasoning. *Yale Law Journal*. 23(1), p. 16-59, 1913.

[10] Institute for Software Research. *xADL 2.0*. University of California, Irvine. http://www.isr.uci.edu/projects/xarchuci/

[11] Institute for Software Research. *ArchStudio 4*. Univ. of Calif, Irvine, 2006.http://www.isr.uci.edu/projects/archstudio

[12] Kuhl, F., Weatherly, R., et al. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice-Hall: Upper Saddle River, New Jersey, 1999.

[13] Medvidovic, N., Rosenblum, D.S., et al. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proc. of the 21st International Conference on Software Engineering (ICSE '99)*. p. 44-53, Los Angeles, CA, May 16-22, 1999.

[14] Meyers, B.C. and Obendorf, P. *Managing Software Acquisition: Open Systems and COTS Products*. Addison-Wesley: New York, 2001.

[15] Nelson, L. and Churchill, E.F. Repurposing: Techniques for Reuse and Integration of Interactive Services. In *Proc. of the Int. Conf. Information Reuse and Integration*. Sep, 2006.

[16] Oreizy, P. *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution*. Thesis (Ph. D., Information and Computer Science), University of California, 2000.http://www.ics.uci.edu/~peymano/papers/thesis.pdf

[17] Rosen, L. *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice-Hall PTR: Upper Saddle River, New Jersey, 2005.

[18] Scacchi, W. and Alspaugh, T.A. Emerging Issues in the Acquisition of Open Source Software by the U.S. Department of Defense. In *Proc. of the 5th Annual Acquisition Research Symposium*. May 13-15, 2008.

[19] St. Laurent, A.M. *Understanding Open Source and Free Software Licensing*. O'Reilly Press: Sebastopol, CA, 2004.

[20] Unity Technologies. End User Lic.Agreement. http://unity3d.com/unity/unity-end-user-license-2.x.html 2008.

[21] Ven, K. and Mannaert, H. Challenges and Strategies in the Use of Open Source Software by Independent Software Vendors. *Info and Software Tech.* 50, p. 991-1002, 2008.

Intellectual Property Rights Requirements for Heterogeneously-Licensed Systems

Thomas A. Alspaugh
Department of Computer Science
Georgetown University
Washington, DC 20057 USA
alspaugh@cs.georgetown.edu

Hazeline U. Asuncion and Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
{hasuncion,wscacchi}@ics.uci.edu

Abstract

Heterogeneously-licensed systems pose new challenges to analysts and system architects. Appropriate intellectual property rights must be available for the installed system, but without unnecessarily restricting other requirements, the system architecture, and the choice of components both initially and as it evolves. Such systems are increasingly common and important in e-business, game development, and other domains. Our semantic parameterization analysis of open-source licenses confirms that while most licenses present few roadblocks, reciprocal licenses such as the GNU General Public License produce knotty constraints that cannot be effectively managed without analysis of the system's license architecture*. Our automated tool supports intellectual property requirements management and license architecture evolution. We validate our approach on an existing heterogeneously-licensed system.*

1. Introduction

Until recently, the norm for licensed software has been that software is used and distributed under the terms of a single license, with all its components homogeneously licensed under a single proprietary or open-source software (OSS) license. It is increasingly common to see *heterogeneously-licensed (HtL) systems*, whose components are not under the same license [8, 18, 20]. For web systems this has become so common that commercial tools for creating such "mashups" have been available for several years [9, 11]. Carefully constrained design, possibly aided by license exceptions from the copyright owners, may enable the resulting system to have a single specific license [8]. Otherwise the system as a whole has no single license, but rather one or more rights that are the intersection of all the component license's rights, and the union of their obligations. An example of a HtL system is the Unity game development tool, whose license agreement lists eleven distinct licenses for its components, in addition to its overall license terms granting the right to use the system [20].

The *intellectual property* (IP) in a system—copyrights, patents, trademarks, trade dress, and trade secrets—is protected and made available through the licenses of the system and its components. IP requirements are expressed in terms of these licenses and the rights and obligations they entail, and include

- the right to use, distribute, sublicense, etc.;
- the component selection strategy (whether limited to specific licenses, or open to "best-of-breed");
- interoperation of systems with specific IP regimes;
- the extent to which the system will be an open architecture (OA); and
- how it is distributed to, constituted by, and (for OA systems) evolved by users.

The IP requirements interact with the system's design-time, distribution-time, and run-time architectures in distinct ways, with the possibility of rights that conflict with other licenses' obligations, obligations that conflict across licenses, and unobtainable rights. The result can be a system that can't legally be sublicensed, distributed, or used, or that involves its developers, distributors, or users in legal liabilities. Of course, some will ignore these legal issues (and anecdotal evidence indicates that many do), but companies and governments cannot afford to. Source code scanning services provided by third-party vendors address only one after-the-fact aspect of this problem. While heuristics exist for managing IP requirements and are used in HtL system development practice, they impose costs, unnecessarily limit the design space, and can result in a suboptimal, unsatisfactory system.

Software licenses and IP rights represent a new class of nonfunctional requirements, and constrain the development of systems with open architectures.

As part of our ongoing investigation of OSS and OA systems, we performed a grounded theory, semantic parameterization analysis of nine OSS licenses [4]. From this analysis and related work on OSS licensing [7, 17, 19], we were able to produce a metamodel for software licenses and for the contexts in which they are applied, and a calculus for license rights and obligations in license and context models. Using them, we calculate rights and obligations for specific systems, identify conflicts and unsupported rights, and evaluate alternative architectures and components and guide choices. We argue that these calculations are needed in developing systems of components whose licenses can conflict, whose design-time, distribution-time, and run-time architectures are not identical, whose component licenses may change through evolution, or for which a "best-of-breed" component strategy is desired. We have validated the approach by encoding the copyright rights and obligations for a group of OSS and proprietary licenses, implementing an architecture tool to automate calculations on the licenses, and applying it to an OSS-OA reference architecture. These models bring into clear relief the knotty constraints produced by interactions among proprietary licenses and reciprocal licenses such as the GNU General Public License (GPL), Mozilla Public License (MPL), and IBM's Common Public License (CPL). We have also discovered a novel second possible mechanism of interaction, through sources shared among compiled components under different licenses.

The main contributions of this work are the concept of a *license firewall* (Section 3.3); a metamodel for software licenses (Section 5); the concept of a *license architecture* (Section 5); an analysis process for determining the rights available for a system and their corresponding obligations (Section 6); an implementation of this analysis in an architecture development environment (Section 7); and the concept of a *virtual license* (Section 8).

The remainder of the paper is organized as follows. Section 2 outlines a motivating example from our own experience. Section 3 gives background. Related work is in Section 4. We discuss our analysis and metamodel of OSS licenses in Section 5, and the system contexts and calculations on them in Section 6. Section 7 presents our tool support and its application to the reference model. We discuss implications in Section 8 and conclude in Section 9.

2. A motivating example

Heterogeneous software licenses can limit architectural choices when building and distributing multi-component systems, as illustrated by our recent experience prototyping a new multimedia content management portal that included support for videoconferencing and video recording and publishing. Our prototype was based on an Adobe Flash Media Server (FMS), and we developed both broadcast and multi-cast clients for video and audio that shared their data streams through the FMS. FMS is a closed source media server whose number of concurrent client connections is limited by a license fee. As the FMS license did not allow for redistribution, we could invite remote users to try out our clients and media services, but we could not offer to share the run-time environment that included the FMS. We could distribute our locally-developed clients and service source code. However, other potential developers at remote locations would then need to download and install a licensed copy of the FMS, and then somehow rebuild our system using the source code we provided and their local copy of the FMS. In our view, this created a barrier to sharing the emerging results from our prototyping effort. We subsequently undertook to replace the FMS with Red5, an open source Flash media server, so we could distribute a run-time version of our content management portal to remote developers. Now these developers could install and use our run-time system, or download the source code, build, and share their own run-time version. Our experience shows how common software R&D efforts can be hampered in surprising ways by software components whose heterogeneous licenses limit distribution and sharing of work in progress.

3. Background

3.1. Intellectual Property (IP)

An individual can own a tangible thing, and have property rights in it such as the rights to use it, improve it, sell it or give it away, or prevent others from doing so, subject to some statutory restrictions. Similarly, an individual can own *intellectual property* (IP) of various types, and have specific property rights in the intangible intellectual property, such as the rights to copy, use, change, distribute, or prevent others from doing so, again subject to some statutory restrictions. In the United States and most other countries, intellectual property is defined by

- *copyright* for a specific original expression of an idea,
- *patent* for an invention,
- *trademark* for a symbol identifying the origin of products,
- *trade dress* for distinctive product packaging, and
- *trade secret* for an idea kept confidential.

Software licenses are primarily concerned with copyrights and patents, and mention trademarks only to restrict a licensee's use of them; licenses rarely discuss trade dress or trade secrets [17]. In this paper we focus on copyright aspects of licenses.

Copyright is defined by Title 17 of the U.S. Code and by similar law in most other countries. It grants exclusive rights to the author of an original work in any tangible means of expression, namely the rights to reproduce the copyrighted work; prepare derivative works; distribute copies; and (for certain kinds of work) perform or display it. Because the rights are exclusive, the author can prevent others from exercising them, except as allowed by "fair use". The author can also grant others any or all of the rights or any part of them; one of the functions of a software license is to grant such rights, and define the conditions under which they are granted.

Copyright *subsists* in the expression of the original work, that is, the rights begin from the moment the work is expressed. In the U.S. a copyright lasts for the author's lifetime plus 70 years, or 95 years for *works for hire* [21].

3.2. Open-Source Software (OSS)

In contrast to traditional proprietary licenses, used by companies to retain control of their software and restrict access and rights to it outside of the company, OSS licenses are designed to encourage sharing of software and to grant as many rights as possible. OSS licenses may be classified as *academic* or *reciprocal*. The academic licenses, including the Berkeley Software Distribution (BSD) license, the Massachusetts Institute of Technology (MIT) license, the Apache Software License (ASL), and the Artistic License, grant nearly all rights and impose few obligations. Anyone can use the software, create derivative works from it, or include it in proprietary projects; typically the obligations are to not remove the copyright and license notices from the software.

Reciprocal licenses encourage sharing of software in a different way, by imposing the condition that the reciprocally-licensed software not be combined (for varying definitions of "combined") with any software that is not then released in turn under the reciprocal license. The goal is to ensure that as open software is improved, by whomever and for whatever purpose, it remains open. The means is by preventing improvements from vanishing behind closed, proprietary licenses. Examples of reciprocal licenses are GPL, MPL, and CPL.

Licenses of both types typically disclaim liability, assert that no warranty is implied, and obligate licensees to not use the licensor's name or trademark. Newer licenses tend to discuss patent issues, either giving a limited patent license along with the other rights, or stating that patent rights are not included.

Several newer licenses add interesting degrees of flexibility. Most licenses grant the right to sublicense under the same license, or in some cases under any version of the same license. IBM's CPL grants the right to sublicense under any license that meets certain conditions; CPL itself meets them, of course, but several other licenses do as well. Finally, 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 standards for OSS licenses, reviews OSS licenses under those standards, and gives its approval to those that meet them [16]. OSI publishes a standard repository of approximately 70 approved OSS licenses.

It has been common for OSS projects to require that developers contribute their work under conditions that ensure the project can license its products under a specific OSS 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 license configuration, in which the rights to a system's components are homogenously granted and the system has a well-defined OSS license, was the norm and continues to this day.

3.3. Open Architecture (OA)

Open architecture (OA) software is a customization technique introduced by Oreizy [15] that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. Almost a decade later, we see more and more software-intensive systems developed using an OA strategy not only with open source software (OSS) components but also proprietary components with open APIs (e.g. [20]). Developing systems using the OA technique can lower development costs [18]. Composing a system with HtL components, however, increases the likelihood of liabilities stemming from incompatible licenses. Thus, in this paper, 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*.

OA may simply seem to mean software system architectures incorporating OSS components and open application program interfaces (APIs). But not all such architectures will produce an OA, since the available license rights of an OA depend on: (a) how and why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the licenses of different OSS components encumber all or part of a software system's architecture into which they are integrated [1, 18].

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed [2].

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 [6] or "mashups" [14]. Each may have its own license.

Executable components—These components are in binary form, and the source code may not be open for access, review, modification, or possible redistribution [17]. If proprietary, they often cannot be redistributed, and so are present in the design- and run-time architectures but not at distribution-time.

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

Application program interfaces/APIs—Availability of externally visible and accessible APIs is the minimum requirement to form an "open system" [13]. 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 [12], 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 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 1 provides an overall view of a reference architecture that includes all the software elements above. This reference architecture has been instantiated in a number of configured systems that combine OSS and closed source components. One such system handles time sheets and payroll at the 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

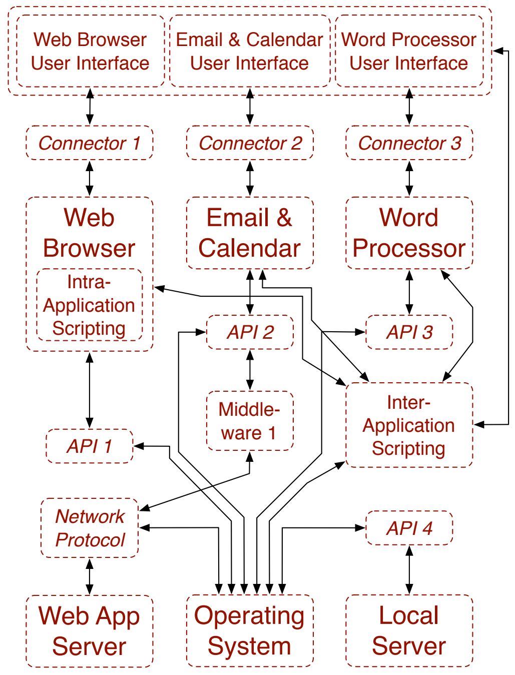


Figure 1. Reference architecture for a heterogeneously-licensed e-business system; connectors (which have no license) are italicized

as an Apache Web server. The 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.

4. Related work

There has been little explicit guidance on how best to develop, deploy, and sustain complex software systems when different OA and OSS objectives are at hand. Ven [22] and German [8] are recent exceptions.

Ven discusses the challenges faced by independent software vendors who develop software using OSS and proprietary components, focusing on the evolution and maintenance of modified OSS components [22].

German models a license as a set of grants, each of which has a set of conjoined conditions necessary for the grant to be given [8]. Interaction between licenses is analyzed by examining pairs of licenses in the context of five types of component connection. He also identify twelve patterns for avoiding license mismatches, found in a large group of OSS projects, and characterize the patterns using their model. Our license model extends German's to address semantic connections between obligations and rights.

Legal scholars have examined OSS licenses and how they interact in the legal domain, but not how licenses apply to specific HtL systems and contexts [7, 19]. For example, Rosen surveys eight existing OSS licenses and creates two more of his own, the Open Source License and the Academic Free License, written to professional legal standards [17]. He examines license interactions primarily in terms of the categories of reciprocal and non-reciprocal licenses, rather than in terms of specific licenses.

Breaux et al. have analyzed regulatory rules in another domain, that of privacy and security [3, 4]. We adapt their approach in our analysis of OSS licenses.

Our previous work examines how best to align acquisition, system requirements, architectures, and OSS elements across different software license regimes to achieve the goal of combining OSS and OA [18].

5. Analyzing software licenses

A particularly knotty challenge is the problem of heterogeneous licenses in software systems. In order to illuminate the specifics of this challenge and provide a basis for addressing it, we analyzed a representative group of common OSS licenses and (for contrast) a proprietary license, using an approach based on Breaux's semantic parameterization [4].

We analyzed these licenses:

1. Apache 2.0
2. Berkeley Software Distribution (BSD)
3. Common Public License (CPL)
4. Eclipse Public License 1.0
5. GNU General Public License 2 (GPL)
6. GNU Lesser General Public License 2.1 (LGPL)
7. MIT
8. Mozilla Public License 1.1 (MPL)
9. Open Software License 3.0 (OSL)
10. Corel Transactional License (CTL)

We obtained the text of the nine OSS license from the Open Source Initiative web site [16], and the text of the proprietary CTL license from Corel's web site [5].

The stages of the analysis were:

1. First we disambiguated forward and backward references, identified synonyms, and distinguished polysemes that expressed different meanings with identical wording. We identified terms of art from copyright law, such as "Derived Work", and specialized terms defined for a particular license, such as "work based on the Program" for GPL and "Electronic Distribution Mechanism" for MPL. From this we constructed (automatically) a concordance to aid us in the remainder of the analysis. The concordance indexed the instances of each distinguished word term, excluding minor words such as articles, conjunctions, and prepositions whose use in a particular license carried no specialized meaning, and tagged each sentence with its section, paragraph, and sentence sequence numbers. Figure 2 shows a portion of the concordance for GPL.
2. Next we identified the parts of each license that had no legal force, such as GPL 2's "Preamble" section, or that dealt with any rights or obligations other than those for copyright, such as patents, trademarks, implied warranty, or liability, iterating with the concordance to confirm the identifications. The remainder of our analysis focused on copyright.
3. Using the concordances across the licenses, and guided by legal work on OSS licenses [7, 17, 19], we identified words and phrases with the same intensional meaning, and textual structures parallel among the licenses. From these we iterated to identify natural language patterns each of which could be used as a restricted natural language statement (RNLS) to express the licenses.

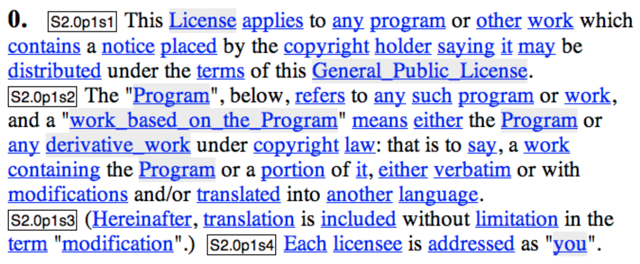

0. S2.0p1s1 This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General_Public_License.
S2.0p1s2 The "Program", below, refers to any such program or work, and a "work_based_on_the_Program" means either the Program or any derivative_work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language.
S2.0p1s3 (Hereinafter, translation is included without limitation in the term "modification".) S2.0p1s4 Each licensee is addressed as "you".

Figure 2. GPL 2 concordance, sect. 2.0 par. 1

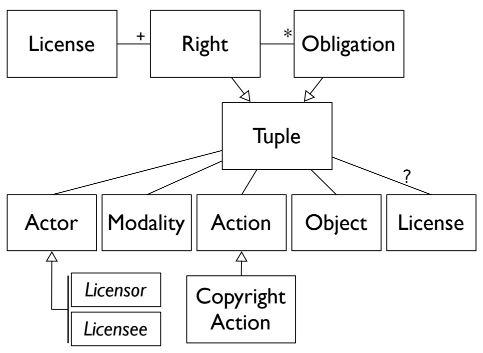
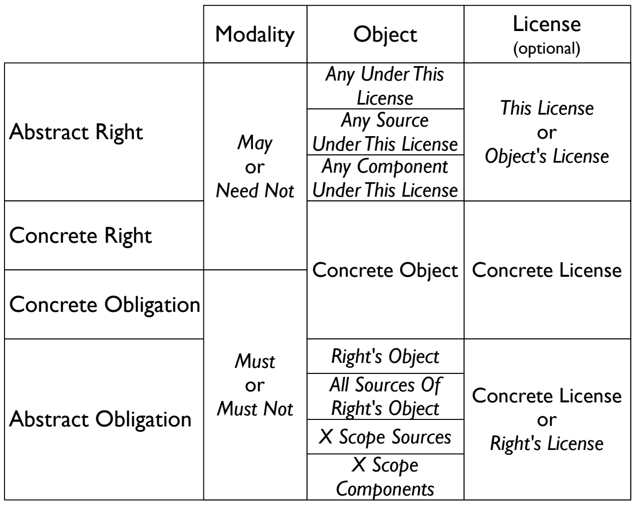


Figure 3. The metamodel for licenses

Our metamodel, derived from the patterns we identified, is shown in Figure 3. A license consists of one or more *rights*, each of which entails zero or more *obligations*. Rights and obligations have the same structure, a tuple comprising an actor (the *licensor* or *licensee*), a modality, an action, an object of the action, and possibly a license referred to by the action.



	Modality	Object	License (optional)
Abstract Right	*May* or *Need Not*	*Any Under This License*	*This License* or *Object's License*
		Any Source Under This License	
		Any Component Under This License	
Concrete Right		Concrete Object	Concrete License
Concrete Obligation	*Must* or *Must Not*		
Abstract Obligation		*Right's Object*	Concrete License or *Right's License*
		All Sources Of Right's Object	
		X Scope Sources	
		X Scope Components	

Figure 4. Modality, object, and license

We found a wide variety of license actions, some of which are defined in copyright law or derived from it and are distinguished as copyright actions. The possible modalities, objects, and licenses are shown in Figure 4.

The RNLS textual form of an example *abstract right*, (one not bound to a specific object) extracted from the BSD license is

> Licensee · may · distribute <Any Source> under <This License>

where "distribute under" is a copyright action and the abstract object <Any Source> quantifies the right over all sources licensed under the license containing the right (here, BSD); an example concrete obligation is

> Licensee · must · retain the [BSD] copyright notice in [`file.c`]

where "retain the copyright notice" is an action that is not a copyright action, BSD is the concrete license the action references, and `file.c` is the concrete object the action references. The RNLS actions are defined with tokens identifying where the tuple's object and (if present) license are inserted, for example in the GPL action "sublicense % under ˆ" which becomes "sublicense *OBJECT* under *LICENSE*". Figure 5 is an informal illustration of how actions may contain concrete objects and licenses, references to objects or licenses bound elsewhere, or quantifiers using the information in the license architecture abstraction described below to produce sets of rights or obligations.

We used the metamodel to express the software licenses and their rights, obligations, and lower level components as Java objects. The constants for the two actors, the four modalities, and the two license quantifiers were

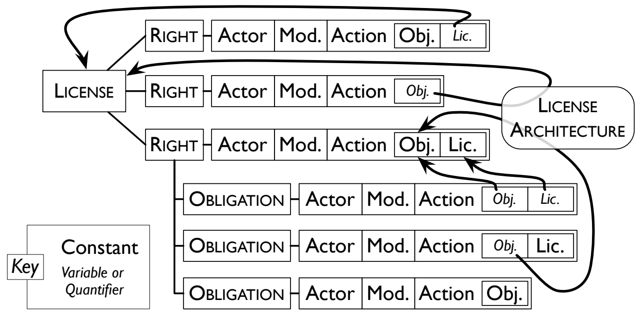


Figure 5. Object/license references, informally

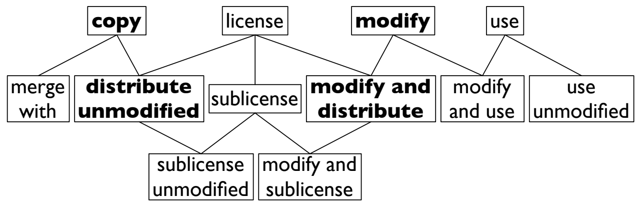


Figure 6. Partial order of copyright actions; actions defined in the Copyright Act in bold

implemented as singleton objects of classes that implemented their semantics. Copyright actions became defined constants of the Action class, while the remaining non-copyright actions were unified if their intensional meanings were identical. From this basis we constructed singleton objects for each license, reusing the same object for each instance of its concept in the licenses.

From our analysis we confirmed that the copyright actions form a partial order, in which a higher copyright implies the rights it is connected to below it.

Figure 6 shows a portion of the copyright partial order, using brief phrases to identify each defined action. The relation defining the order is "implied by". For example, "copy" is implied by "sublicense unmodified", since it accomplishes nothing to sublicense copies without making copies, so if we are obligated to sublicense an unmodified component but fail to have the right to copy it, we cannot meet our obligation and do not have whatever rights demand it. The copyright actions are the ones specifically mentioned in the Copyright Act [21]; we incorporated all actions appearing in the licenses we analyzed into the full ordering.

This model of licenses gives a basis for reasoning about licenses, applying them to actual systems, and calculating the results. The additional information we need about the system is defined by the list of quantifiers that can appear as objects in the rights and obligations. The information

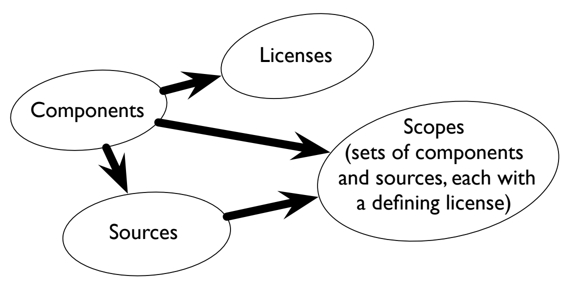

Licenses
Components
Scopes (sets of components and sources, each with a defining license)
Sources

Figure 7. The license architecture metamodel

needed is the *license architecture* (LA), 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 (Figure 7).

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

We note that the obligation quantifiers we identified in OSS licenses include ones that can set up conflicts through the license scopes, as is well known for GPL and other reciprocal licenses. However, we also identified an obligation quantifier over all sources of a component, and note that it raises the possibility of a conflict arising through components that share a source. We believe this conflict path is novel, and are investigating in what contexts, if any, it could occur.

6. Analyzing license architectures

In order to make calculations about the rights and obligations for a specific system, we iterate over its components, instantiating each component's license with the component's information. From the resulting concrete rights and obligations, we can determine the set of rights available for the system as a whole, and the set of concrete obligations that must be met in order to get those rights.

The instantiation proceeds conceptually as follows.

Each of the abstract rights in every license has as its object either "Any Under This License", "Any Source Under This License", or "Any Component Under This License". An abstract right *R* in license *L* is made into one or more concrete rights by replacing "Any Component" with each component licensed under *L* in succession, "Any Source" similarly with sources, and "Any" with either. If the abstract right *R*'s license is "Object's License", then in each concrete right *r* the license is replaced by *r*'s object's license.

Each of *R*'s obligations *O* is made into one or more concrete obligations *o* for each *r*. If *O*'s object is "Right's Object", then there will be a single *o*, and *r*'s object is used as its object; if *O*'s object is "All Sources Of Right's Object", then there will be an *o* for each source *s* of *r*'s object (which must be a component), and that *o*'s object will be *s*; if *O*'s object is "L' Scope Components" for some license L', then there will be an *o* for each component *c* in *r*'s object's scope, under the definition of "scope" in L', and that *o*'s object will be *c*; and if *O*'s object is "L' Scope Sources" then analogously for each such component's sources.

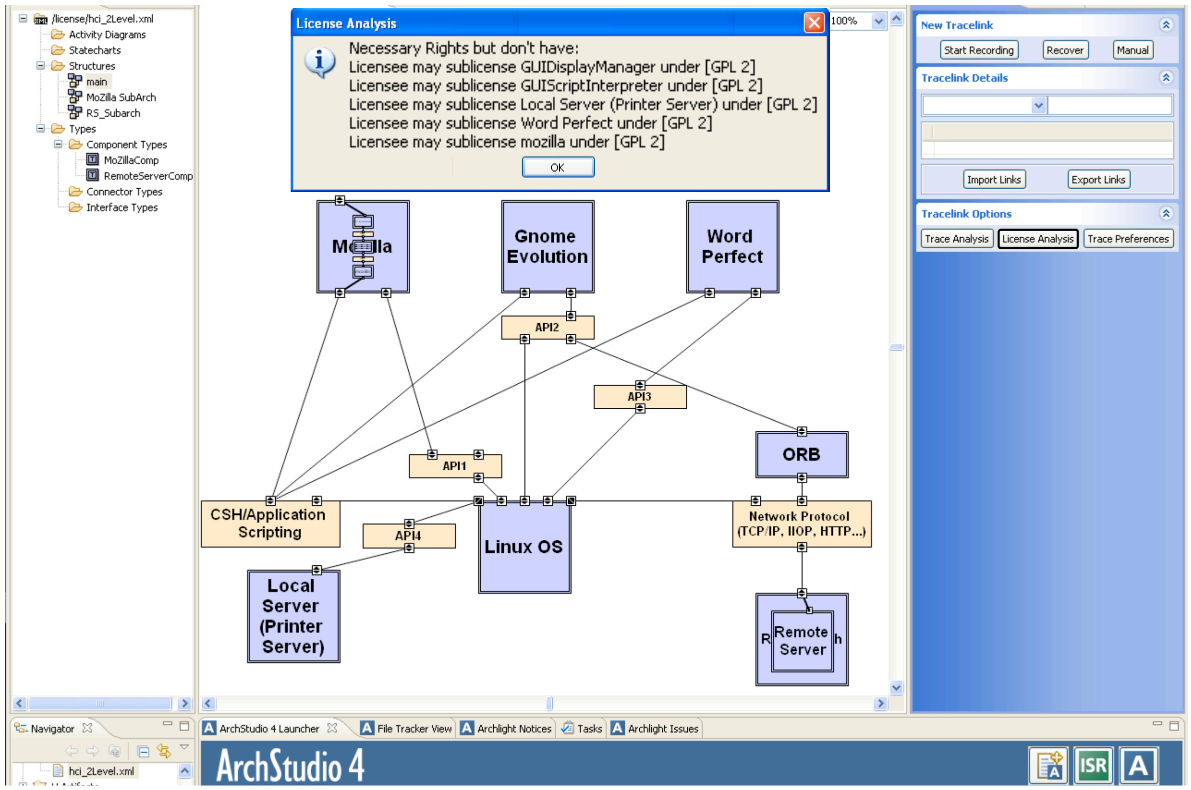
However, we do not yet know how to fulfill these concrete obligations. We generate the *correlative right* from the correlate of the obligation's modality ("may" for "must", "need not" for "must not") and the remaining parts of the obligation. If the correlative right is a copyright right, we must use its object's license to fulfill it: we seek the license of this right's object, find an abstract right that generalizes this right, and iterate the process for this abstract right and the parts of the correlative right. If there is no such right in the license, we iterate it again for the right's successive containing rights in the partial order of copyright rights, hoping to get all the rights that include it. If not, we save the correlative right as an *unfulfilled correlative right*.

If the correlative right is not a copyright right, we do not have to obtain it through a license. However, we must still check whether it conflicts with an obligation. After all the obligations have been determined, we compare it against them, looking for the *opposite obligation*, the obligation that matches the right's actor, action, object, and modality, but has the opposite modality ("must not" for "may", "must" for "need not"). If we find such an obligation, then there is a *right-obligation conflict*.

Finally, we go through the concrete obligations looking for pairs of obligations identical except for their modalities. A pair of such obligations indicates an *obligation-obligation conflict*.

The presence of any unfulfilled correlative rights, right-obligation conflicts, or obligation-obligation conflicts indicates that the full set of license rights can't be achieved for all the components in the system. However, we may not need the full set of rights. More commonly we need a subset of the copyright rights, for example the rights to use and distribute the system. This is equivalent to having that set of rights for each component of the system, determined by examining each component's rights for the desired rights.

If rights are those desired, then the collected concrete



License Analysis

Necessary Rights but don't have:
Licensee may sublicense GUIDisplayManager under [GPL 2]
Licensee may sublicense GUIScriptInterpreter under [GPL 2]
Licensee may sublicense Local Server (Printer Server) under [GPL 2]
Licensee may sublicense Word Perfect under [GPL 2]
Licensee may sublicense mozilla under [GPL 2]

OK

Figure 8. The license architecture analysis tool identifying unavailable rights

obligations for then forms the basis for automated testing that IP obligations have been met [18].

The process outlined above is not one that developers would be keen to follow manually. In the next section we discuss an approach for automating it, both to examine its potential usefulness as an HtL system development tool and to validate the analysis process.

7. Automating the analysis

We implemented the analysis tool within the traceability view of the ArchStudio software architecture environment [10]. We annotated the xADL software architecture model with component licenses and sources and used subarchitectures to model scopes. This approach provides:

- The ability to model software systems and specify the corresponding licenses at different levels of granularity. We provide the option of specifying licenses at a fine-grained level, for example licenses assigned to components at the level of a single Web service, such as the Google Desktop Query API, or at a coarse-grained level, for example one license assigned to a set of services provided by Google Desktop APIs `http://code.google.com/apis/desktop/`.
- The ability to model software systems at different architecture levels and to analyze license interactions across the different architecture levels. For instance, if a sub-subsystem *X* contains heterogeneous licenses and is itself part of a bigger system *Y* with heterogeneous licenses, our approach is able to analyze license interactions between sub-subsystem *X* and System *Y*. We expect to analyze license interactions across multiple architecture levels.
- The modeling approach maps well to the way real software systems are configured.
- Automated license analysis is informed by the additional knowledge of the system configuration. This is one of our contributions beyond current techniques and approaches. Simply modifying the system configuration can result in different sets of available rights or required obligations. Thus, the same set of components may be analyzed with or without specific license firewalls inserted among them.

Figure 8 shows an analysis of a design that specializes the reference architecture in Figure 1.

Scalability is always an issue for any approach. We conclude that our initial algorithm is quadratic in the number of components with licenses, which for architectures of up to several hundred components is manageable. The approach requires modeling the system architecture, in common with many other research approaches, and annotating it to produce the license architecture, which we feel is a worthwhile tradeoff for developers following a best-of-breed strategy or who need to manage reciprocal and proprietary components or design-, distribution-, and run-time architectures that differ in significant ways.

The integration of the analysis with architecture design and evaluation supports easy management of licenses across the software development lifecycle and across product variations. For instance, as the software evolves, analysts may consider replacing a proprietary word processing component with an OSS component. By simply modifying the architecture model and running the automated license analysis, the analyst learns the new set of rights and obligations. Similarly, an analyst can create product variations to suit a particular deployment platform or customer IP requirements. These product variations can be stored with ArchStudio and retrieved or analyzed at any later time.

8. Discussion

Our efforts in this study are motivated in part by a desire to understand how best to accommodate the development of complex software systems whose components may be subject to different IP licenses. These licenses stipulate the rights and obligations that must be ensured. However, system composition can incorporate components with different licenses at architectural design time, at distribution time, or at installed release run time. Thus, we must consider what overall license schemes we can accommodate, as well as identify the consequences (freedoms and constraints) each scheme can realize.

There are at least two kinds of software license/IP schemes that impose requirements on how software systems will be developed: (a) a single license for the complete software system, and (b) a heterogeneous license scheme of rights and obligations for the complete system incorporating components with different licenses. We consider each in turn.

A single license scheme—There is often a desire to specify a single license at architecture design time in order to insure a composed software system with single license compatible scheme at distribution time, and also at run time. Software licenses like GPL encourage this as part of their overall IP strategy for insuring software freedom. Similarly, there is desire to determine whether a single known license can cover a designed or released system [8]. However, a single license regime cannot in general be guaranteed to occur by chance; instead it is most effectively determined by design. In either case, it must be specified as a nonfunctional requirement for software development. But satisfying such a requirement limits the choice of software components that can be included in the system design and the system composition at distribution- and run-time to those compatible (or subsumed) with the required overall system license. Consequently, our goal in this case is to insure a simple, homogeneous scheme relying on known licenses to determine the propagation and enforcement of their constraints.

A heterogeneous license scheme—In contrast to a single license scheme, a heterogeneous license scheme allows a software system to incorporate components with different IP licenses. Such a scheme gives more degrees of freedom than a single license scheme. For example, it allows for best-of-breed component selection, considering components with a range of licenses rather than only those with a specific license. It also allows for specification and design of software systems conforming to a reference architecture [2]. This enables a higher degree of software reuse through inclusion of reusable software components that have a substantial prior investment in their development and use. Similarly, when relying on a reference architecture, design-time component choices need not be encumbered by license constraints, since the resulting system license rights and obligations need only be determined at distribution-time and run-time. Furthermore, the distribution- and run-time system compositions are not limited to a single license; instead they are constrained only by the license rights and obligations that ensue for the entire system.

In a heterogeneous license scheme, the overall system rights and obligations can form a virtual license—a license whose rights and obligations can be determined, tested, and satisfied at any time, without being a previously approved license type, e.g. via the OSI license approval scheme [16].

This enables prototyping both software system compositions and new software license types, and determining their effect when later mixed with existing software components or licenses. However, determining the scope of rights and obligations in an overall composed system will be challenging without an automated tool such as the one we demonstrated.

Overall, the key observation is that there is a choice of ways to proceed in terms of guidance both for those who seek a single license regime for all components and system compositions, as in GPL-based software, and for those who seek to work with multiple software component licenses in order to develop the best possible system designs they can realize.

Finally, it now appears possible to design a pure software IP requirements analysis tool whose purpose is to reconcile the rights and obligations of different licenses, whether new or established. Such a tool will not depend on specific software architectures or distributions for analysis. It may be of value especially to legal scholars or IP lawyers who want to design or analyze alternative IP rights and obligations schemes, as well as to software engineers who want to develop systems with assurable IP rights and obligations. That tool is beyond the scope of our effort here. But it is noteworthy that such a tool can emerge from careful analysis of the requirements for open architecture software systems of HtL components.

9. Conclusion

Software licenses and IP rights represent a new class of nonfunctional requirements that increasingly constrain the development of heterogeneously-licensed systems. There has been no model to support analysis and management of these requirements in the context of specific systems, and the heuristics used in practice to deal with them unnecessarily limit the design and implementation choices. It has not been possible to follow a best-of-breed strategy in selecting components without unduly constraining architectural decisions.

In this paper we have presented a metamodel for software licenses through which they can be made the bases of IP rights calculations on system architectures. We defined the concepts of a license firewall, license architecture, and virtual license, providing a basis for analyzing and understanding the issues involved in HtL system IP rights. We outlined an algorithm for calculating concrete rights and obligations for system architectures that identifies conflicts and needed rights, and validated the metamodel and algorithm by formalizing licenses, incorporating model, licenses, and algorithm into the ArchStudio environment, and calculating IP rights and obligations for an existing reference architecture and an instantiation.

Future work includes abstracting our approach to work with licenses in the absence of specific systems, extending it to patent aspects of OSS licenses, and applying it to the challenges of software acquisition.

Acknowledgments

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

The authors thank Paul N. Otto for reviewing the discussion of IP law; all remaining faults are due to the authors.

References

[1] T. A. Alspaugh and A. I. Antón. Scenario support for effective requirements. *Inf. and Softw. Tech.*, 50(3):198–220, Feb. 2008.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

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

[4] T. D. Breaux, A. I. Anton, and J. Doyle. Semantic parameterization: A process for modeling domain descriptions. *ACM Trans. on Softw. Eng. and Meth.*, 18(2), 2008.

[5] Corel Transactional License, 2008. `http://apps.corel.com/clp/terms.html`.

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

[7] R. Fontana, B. M. Kuhn, E. Moglen, M. Norwood, D. B. Ravicher, K. Sandler, J. Vasile, and A. Williamson. *A Legal Issues Primer for Open Source and Free Software Projects*. Software Freedom Law Center, 2008.

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

[9] D. Hinchcliffe. Assembling great software: A round-up of eight mashup tools, Sept. 2006.

[10] Institute for Software Research. ArchStudio 4. Technical report, University of California, Irvine, 2006. `http://www.isr.uci.edu/projects/archstudio/`.

[11] C. Kanaracus. Adobe readying new mashup tool for business users. *InfoWorld*, July 2008.

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

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

[14] L. Nelson and E. F. Churchill. Repurposing: Techniques for reuse and integration of interactive systems. In *Int. Conf. on Information Reuse and Integration (IRI-08)*, page 490, 2006.

[15] P. Oreizy. *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution*. PhD thesis, University of California, Irvine, 2000.

[16] Open Source Initiative, 2008. `http://www.opensource.org/`.

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

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

[19] A. M. St. Laurent. *Understanding Open Source and Free Software Licensing*. O'Reilly Media, Inc., 2004.

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

[21] U.S. Copyright Act, 17 U.S.C., 2008. `http://www.copyright.gov/title17/`.

[22] K. Ven and H. Mannaert. Challenges and strategies in the use of open source software by independent software vendors. *Inf. and Softw. Tech.*, 50(9-10):991–1002, 2008.

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 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.

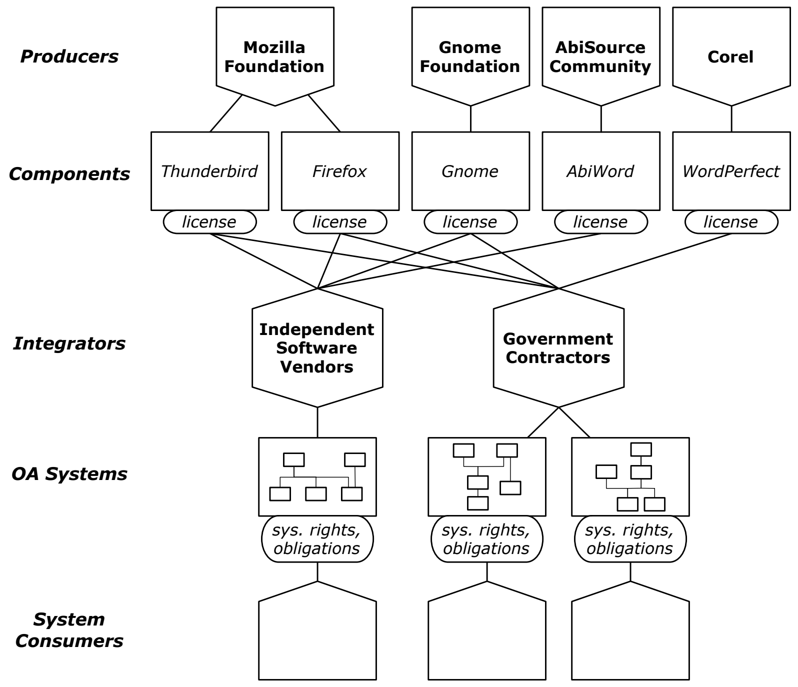


Fig. 1. A hypothetical ecosystem in which OA systems are developed

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. Producers may manage their evolution with a loose coordination among releases, for example as between the Gnome and Mozilla organizations. Releases of producer components create a tension through the ecosystem relationships with the 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 been instantiated in a number of configured systems that combine OSS and closed source components. One such system handles time sheets and payroll

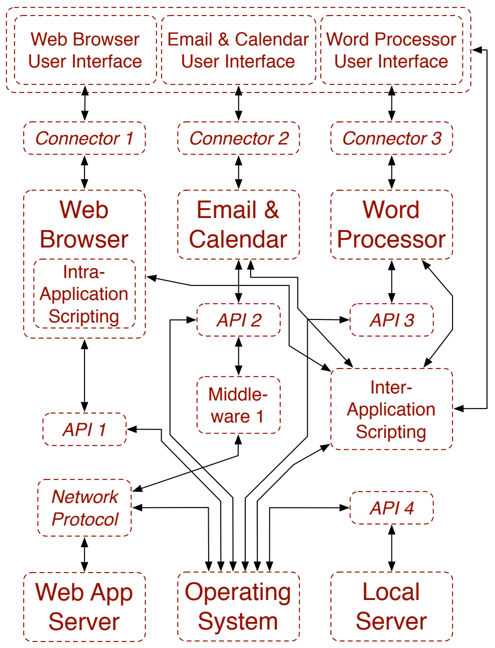


Fig. 2. Reference architecture for a heterogeneously-licensed e-business system; connectors (which have no license) are italicized

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 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).

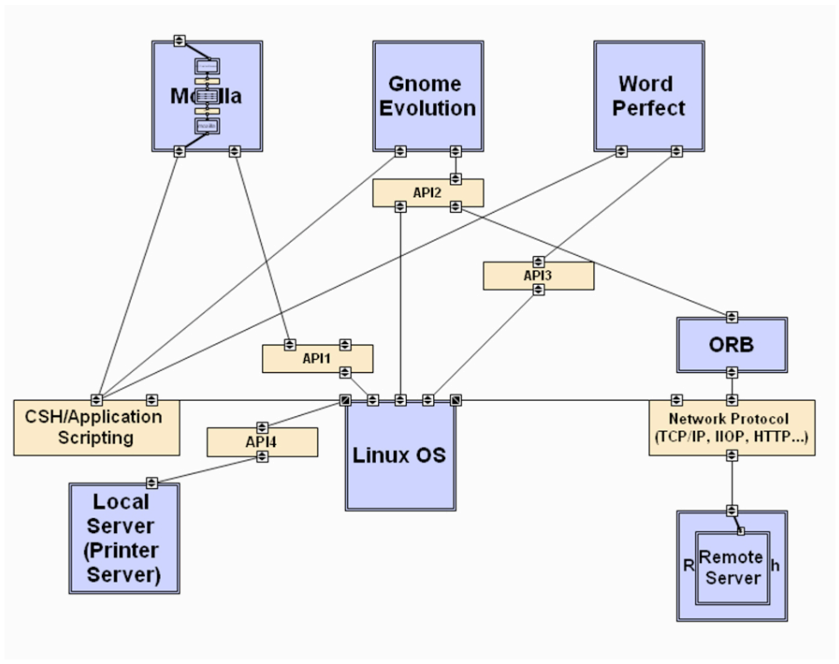

Mozilla
Gnome Evolution
Word Perfect
API2
API3
ORB
API1
CSH/Application Scripting
API4
Linux OS
Network Protocol (TCP/IP, IIOP, HTTP...)
Local Server (Printer Server)
Remote Server

Fig. 3. Instance architecture for a heterogeneously-licensed e-business system

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 complexm 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 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

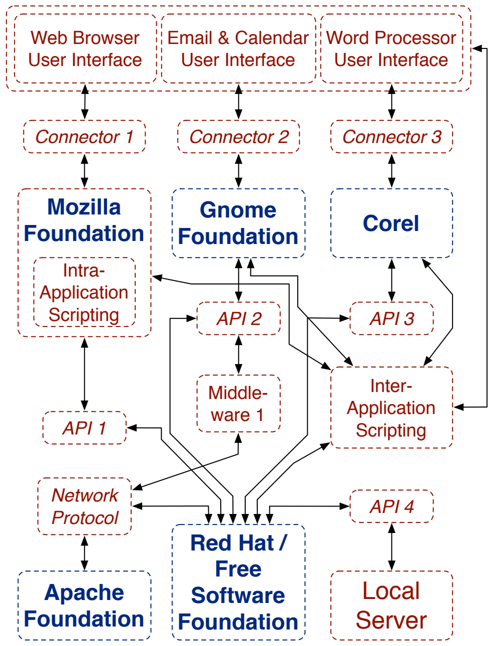

Web Browser User Interface
Email & Calendar User Interface
Word Processor User Interface
Connector 1
Connector 2
Connector 3
Mozilla Foundation
Intra-Application Scripting
Gnome Foundation
Corel
API 2
API 3
Middle-ware 1
Inter-Application Scripting
API 1
Network Protocol
API 4
Red Hat / Free Software Foundation
Apache Foundation
Local Server

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

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 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

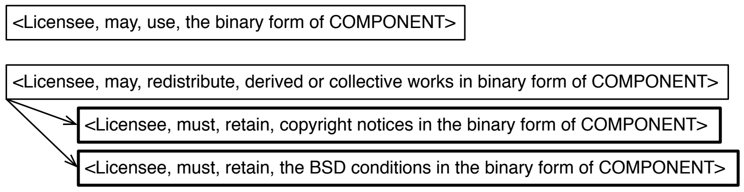


Fig. 5. Tuples for some rights and obligations of the BSD license

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.

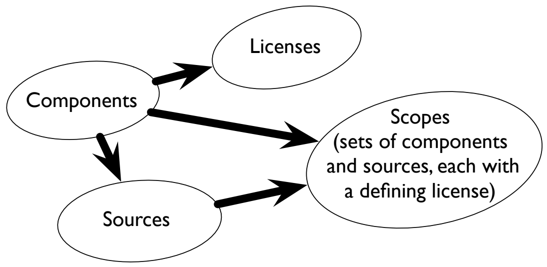


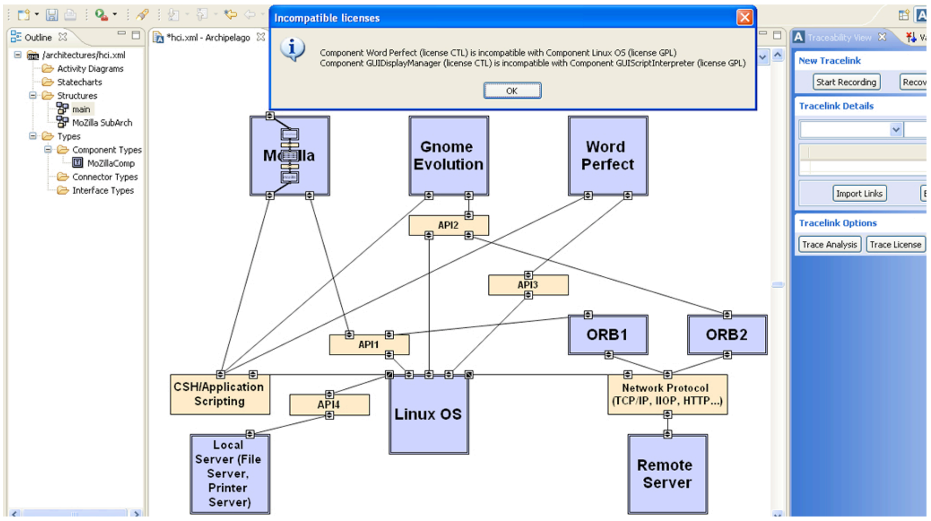
Fig. 6. The license architecture metamodel

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


Incompatible licenses

Component Word Perfect (license CTL) is incompatible with Component Linux OS (license GPL)
Component GUIDisplayManager (license CTL) is incompatible with Component GUIScriptInterpreter (license GPL)

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)