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 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.

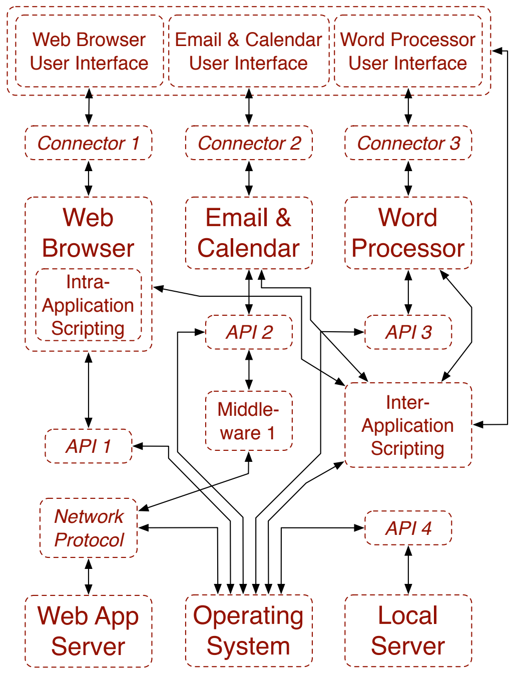


Figure 1. Reference architecture for a heterogeneously-licensed e-business system; connectors (which have no license) are italicized

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.

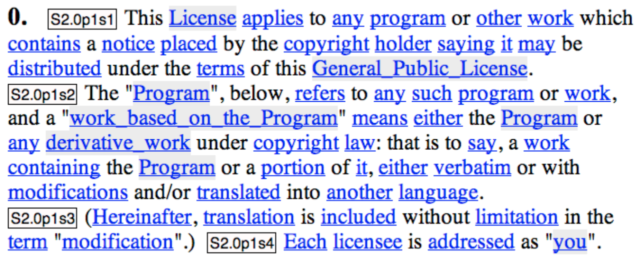

0. S2.0p1s1 This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General_Public_License. S2.0p1s2 The "Program", below, refers to any such program or work, and a "work_based_on_the_Program" means 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

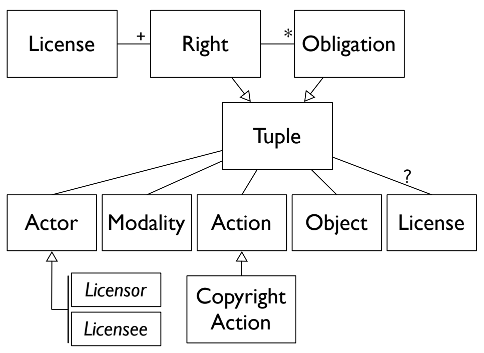


Figure 3. The metamodel for licenses

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

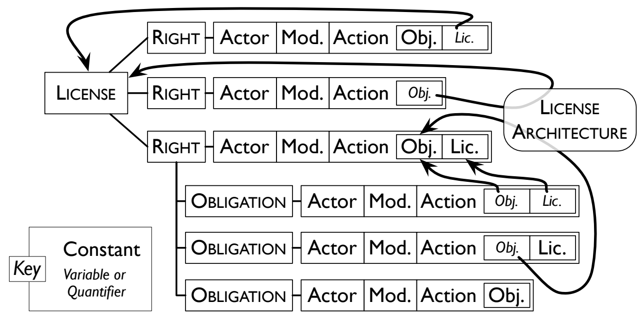


Figure 5. Object/license references, informally

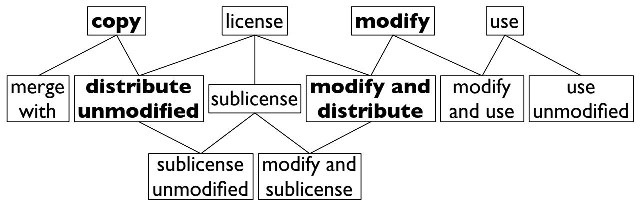


Figure 6. Partial order of copyright actions; actions defined in the Copyright Act in bold

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

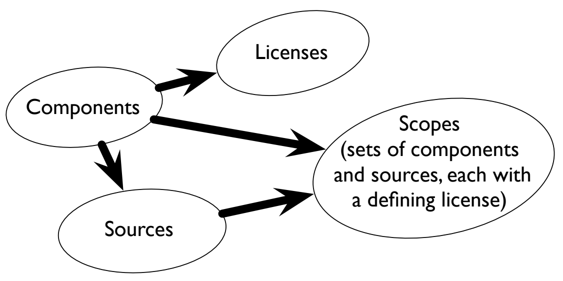

Licenses
Components
Scopes (sets of components and sources, each with a defining license)
Sources

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.

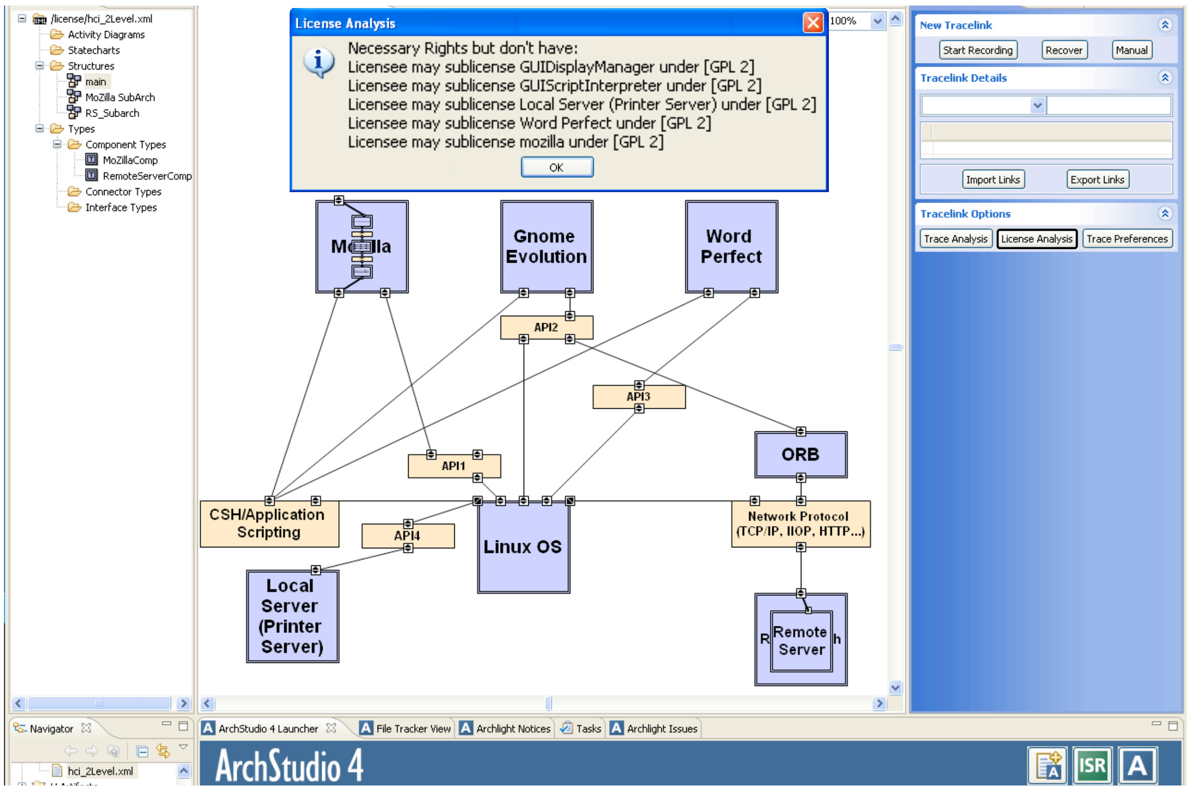
However, we do not yet know how to fulfill these concrete obligations. We generate the *correlative right* from the correlate of the obligation's modality ("may" for "must", "need not" for "must not") and the remaining parts of the obligation. If the correlative right is a copyright right, we must use 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



```
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]
```

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.