Presenting Software License Conflicts through Argumentation

Thomas A. Alspaugh
Computer Science Dept.
Georgetown University
Washington, DC, USA
thomas.alspaugh@acm.org

Hazeline U. Asuncion
Computing and Software Systems
University of Washington, Bothell
Bothell, Washington, USA
hazeline@u.washington.edu

Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, California, USA
wscacchi@ics.uci.edu

***Abstract*—Heterogeneously-licensed systems pose new challenges to architects and designers seeking to develop systems with appropriate intellectual property rights and obligations. In the extreme case, license conflicts may prevent a system's legal use. Our previous work showed that rights, obligations, and conflicts can be calculated. But architects benefit from fuller information than simply (for example) a list of conflicts. In this work we demonstrate an approach for presenting intellectual property results in terms of arguments supporting them. The network of argumentation provides not only an explanation of each conclusion, but also a guide to the tradeoffs available in choosing among design alternatives with different licensing results. The approach has been integrated into the ArchStudio software architecture environment. We present an illustrative example of its use.**

I. Introduction

An increasing number of development organizations are adopting a strategy in which software-intensive systems are composed of *heterogeneously licensed* (HtL) components, with different components governed by different software licenses. The components are either open source software (OSS) or proprietary software with open application programming interfaces (APIs), and are combined in an open architecture (OA) in which components with comparable interfaces can be substituted for each other [10]. Under this strategy the development organization becomes an integrator of components largely produced elsewhere, interconnected to achieve the desired result.

The resulting OA systems can achieve reuse benefits such as reduced costs, increased reliability, and potentially increased agility in evolving to meet changing needs. But rather than a single proprietary license as when acquired from a proprietary vendor, or a single OSS license as in uniformly-licensed OSS projects, the resulting system typically has no recognized single software license. Instead it has, strictly speaking, a *virtual license* [2] composed of each component's rights and obligations for that component under its governing license. The rights available for the system as a whole are the intersection of the rights sets for each component. In some cases the licenses may produce conflicting obligations and this intersection is empty, leaving a system that cannot legally be used, distributed, or modified. An emerging challenge is to realize the reuse benefits of HtL systems while managing virtual licenses to ensure that the desired system rights are available for an acceptable set of obligations.

In our previous work (summarized in Section IV) we described and implemented a novel approach for calculating conflicting obligations, unavailable rights, and virtual licenses in an architectural design context. Calculation is necessary because the number of entailments in a typical HtL system is large, the system's architecture is constantly evolving, its design-, distribution-, and run-time architectures are often distinct, component licenses evolve and components are relicensed, and the consequences of infringement can be substantial. Therefore identifying conflicts and virtual licenses through calculation is a substantial boon. But we soon realized that *explaining* them was of even greater value.

We present an approach in which arguments are used to explain the results of right and obligation calculations. The calculations proceed by elaborating a directed acyclic graph (dag) of inferences among rights to obligations for entities in the system architecture. In this work we reimplemented the software that performs the calculations so that the dag is retained in its entirety as the primary calculation product, containing within it the obligation conflicts, unavailable rights, and virtual license for the system under analysis. Then an explanation for a specific result corresponds to the traversal of a path through the dag, starting at the result in question and continuing until the question has been answered.

- *Conflicting obligations:* the traversal branches for each obligation to show the desired rights, license provisions, and architectural entities from which that obligation is produced, and at the root of the traversal shows in what ways the obligations conflict.
- *Unavailable rights:* for each such right, a traversal identifies the exclusive copyright right that subsumes the right in question, the architectural entity to which the right pertains, and why no right in the entity's license grants the right in question.
- *Virtual license:* traversals show the chains of inference by which each right and obligation is entailed by the system architecture, the stated license for each component, and the desired rights for the system as a whole.

The dag calculation algorithm follows the steps of legal

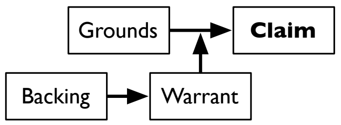


Fig. 1. A *claim*, supported by *grounds*, their pertinence to the claim justified by a *warrant*, whose validity is supported by *backing* (diagram after [14])

reasoning (formalized to support automation) by which an informed analyst would reason out the results. Thus the traversals follow inference paths that follow (in more detail) the paths by which an analyst reasons out the same conclusions.

II. Related Work

The most influential approach for structuring legal arguments is that of Toulmin, who classified the parts of arguments into claims, grounds, warrants, backing, qualifiers, and rebuttals, in a recursive structure with a diagrammatic notation outlined in Figure 1 [14]. His approach has spread beyond the area of legal arguments and is used in general rhetoric and computer science. Toulmin divides arguments into

1) *claims* asserted to be true;
2) for each claim whose truth is disputed, one or more *grounds* supporting it;
3) if it is disputed whether a claim's grounds suffice for it, then a *warrant* stating why the grounds entail the claim;
4) if the warrant is disputed, then *backing* supporting it.

If a ground or backing is disputed, then it is made the claim of a lower-level argument constructed in its support. The recursion of arguments continues as long as grounds or backings are in dispute, or until the original claim is abandoned. (Qualifiers and rebuttals address the degree of strength of arguments, and are not used in the present work.)

Hohfeld sought a theory by which to resolve the imprecise terminology and ambiguous classifications he found in use for legal relationships. In a seminal article published in 1913 and cited to the present day, he set forth a system of eight jural relations intended to express and classify all legal relationships between people. The first four regulate ordinary actions and are *right* ("may"), *no-right* ("cannot"), *duty* ("must"), and *privilege* ("need not"). Each relation has an *opposite* relation whose sense is its opposite, and a *correlative* relation whose sense is its complement. We use Hohfeld's first four jural relations as the basis of our representation of the enactable, testable provisions of software licenses (Section IV).

There has been much work on analysis of laws in AI over the past few decades. A widely-cited example is Sergot et al.'s re-expression of the British Nationality Act as a Prolog program; the resulting program applied the Act to any person's situation and characteristics to determine nationality [12].

A number of researchers have used argumentation to guide decision making, notably Haley et al. who propose an approach for using satisfaction arguments to evaluate and guide evolution of security requirements [7]. Decision choices for which no convincing argument is found are set aside in favor of choices for which stronger arguments have been identified.

III. Licensing Background

A. 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.

Software licenses are primarily concerned with copyrights Copyright is defined by Title 17 of the U.S. Code and by similar law in many 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;
- distribute copies;
- prepare derivative works;
- distribute copies of derivative works; and
- (for certain kinds of work) perform or display it.

Because the rights are exclusive, an author can prevent others from exercising them, except as allowed by "fair use", or can 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.

B. Software Licenses

Traditional proprietary licenses allow a company to retain control of software it produces, and restrict the access and rights that outsiders can have. OSS licenses, on the other hand, encourage sharing and reuse of software, and grant access and as many rights as possible.

Academic OSS licenses such as the Berkeley Software Distribution (BSD) license, the Apache Software License, and perl's Artistic License [1] grant nearly all rights and impose few obligations. Typical academic license obligations are simply to not remove the copyright and license notices.

Reciprocal OSS licenses impose an obligation that distributed modifications of reciprocally-licensed software be freely licensed under the same license. Examples are the Lesser General Public License (LGPL), Mozilla Public License (MPL), and Common Public License [1].

Some reciprocal licenses additionally require that software combined with the licensed software (for various definitions of "combined") also be freely licensed under the same license. We term such licenses *propagating*; they are also known as *strong copyleft* licenses. Examples are the General Public License versions 2 and 3 (GPLv2, GPLv3) [1].

Some OSS is *multiply-licensed*, or distributed under two or more licenses. The MySQL database software is distributed either under GPLv2 for OSS projects or a proprietary license for commercial projects. The Mozilla Disjunctive Tri-License licenses the core Mozilla components under any of three licenses (MPL, GPL, or LGPL).

C. *Licenses and Software Architectures*

Certain classes of architectural features affect the application and propagation of license provisions. The most common such features are listed below. A software architecture is composed of components, each of which is a "locus of computation and state" in a system, and connectors which link them and mediate interactions between them.

Software source code components—These can be

- standalone programs,
- libraries, frameworks, or middleware,
- inter-application script code such as C shell scripts, or
- intra-application script code, to creating Rich Internet Applications using domain-specific languages like XUL for the Firefox Web browser [6] or "mashups"[9].

The distinguishing characteristic of a source code component is that its source code is available and it can be modified and rebuilt. Each may have its own explicit license, though often script code connecting programs and data flows has no stated license unless the script is substantial or proprietary.

Executable components—These components are in binary form, with source code not available for access, review, modification, or possible redistribution [11]. 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.

APIs—These are not and cannot be licensed, but connections through APIs can be used to limit the propagation of some license obligations.

Software connectors—These are software elements providing a standard or reusable way of communication through common interfaces, such as High Level Architecture, CORBA, or Enterprise Java Beans. Connectors can also limit the propagation of some license obligations.

Methods of composition—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. Methods of composition affect license obligation propagation, with different methods affecting different licenses. How and to what extent this occurs have not been resolved in court or in practice [5], [13].

Configured system or subsystem architectures—These are software systems used as atomic components of a larger system. Their internal architecture may contain subcomponents under several licenses, which may affect the rights and obligations for the configured (sub)system and the overall system containing it. To minimize license interaction, a configured system or subsystem architecture may be surrounded by what we term a *license firewall* [2], namely a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of obligations.

D. *Heuristics for Designing HtL Systems*

HtL system designers have developed heuristics to guide architectural design while avoiding some license conflicts.

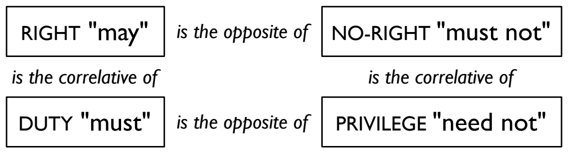


Fig. 2. Hohfeld's four basic relations

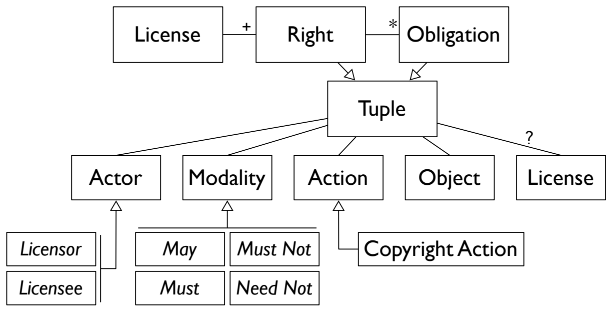


Fig. 3. Metamodel for software licenses

First, it is possible to use a reciprocally-licensed component through a license firewall that limits the scope of reciprocal obligations for specific licenses (depending on how the license provisions are interpreted). Rather than connecting conflicting components directly through static build-time links, the connection is made through a dynamic link, client-server protocol, license shim, or run-time plug-in.

A second approach used by a number of large organizations is to avoid using any components with reciprocal licenses.

Even using design heuristics such as these, keeping track of license rights and obligations across components that are interconnected in complex OAs quickly becomes cumbersome. Organizations wishing to follow a "best-of-breed" component selection policy, without regard to component licenses, face even steeper challenges. Automated support is needed to manage this multi-component, multi-license complexity.

IV. License Rights and Obligations

In our previous work [2] we developed an approach for expressing software licenses that is more formal and less ambiguous than natural language, and that allows us to calculate rights and obligations for an HtL system and identify conflicts arising from the rights and obligations of two or more component's licenses. Our approach is based on Hohfeld's eight fundamental jural relations [8], of which we use *right* ("may"), *duty* ("must"), *no-right* ("must not"), and *privilege* ("need not") (Figure 2). Each relation has a *correlative* relation, which in our context relates an obligation to its necessary right:

- if actor A must perform action X, then A requires the correlative right to perform it, expressed as "A may X";
- if actor A must not perform action X, then A requires the correlative right to not perform it, "A need not X".

We express rights and obligations as tuples (Figure 3):

<actor, modality, action, object, license>

The actor is either the "Licensee" or in a few cases "Licensor" for all the enactable, testable provisions of the licenses

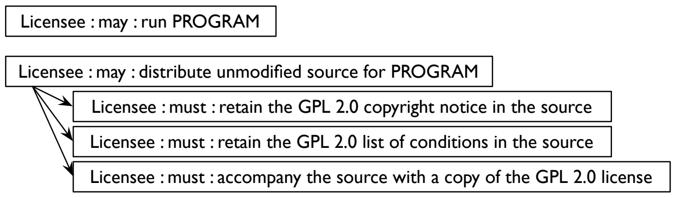

Licensee : may : run PROGRAM
Licensee : may : distribute unmodified source for PROGRAM
Licensee : must : retain the GPL 2.0 copyright notice in the source
Licensee : must : retain the GPL 2.0 list of conditions in the source
Licensee : must : accompany the source with a copy of the GPL 2.0 license

Fig. 4. Some tuples for the GPLv2 license

we have examined [3]. The modality is "may" or "need not" for a right and "must" or "must not" for an obligation. The action is a verb phrase acting on an object, describing what may, need not, must, or must not be done. The object is a module of the system or a related artifact such as a source file, the original version, documentation, and so forth. Typically a license right applies to any of a class of objects distributed under the license, such as any binary file or any modified source file; and the right's obligations will apply to the same object or a related object, such as the right's object's sources or the right's object's originals. For this reason we term rights and obligations as expressed in a license *abstract*, in contrast to a *concrete* right or obligation for one specific entity. Some actions are parameterized by a license as well.

Because copyright rights are exclusive to the copyright holder and licensees, the actions in copyright rights are distinguished from other actions; rights with those actions are only available through the object's license. Rights formed from all other actions are freely and immediately available, unless the object's license obligations restrict them.

A license is expressed as a set of rights, each right associated with zero or more obligations that must be fulfilled be granted it, and possibly a set of overall obligations that must be fulfilled for the license as a whole. Figure 4 sketches two rights from GPL version 2.0 (GPLv2), the first with no obligations and the second with three corresponding obligations.

The details of the license specification approach are described in our earlier work [2], [3].

V. Applying Licenses to Software

A. Calculating the Inference Dag

In order to obtain a particular desired right r for a specific module or other entity e, in other words a desired *concrete right*, one of two cases must hold:

1) r is *not* subsumed by any of the five copyright rights, and does not conflict with any general obligation of r's license L. In this case r is freely available.
2) r *is* subsumed by an abstract right R of the license, with e likewise subsumed by R's object. In this case all R's obligations $O_1, O_2, \ldots, O_n$ must be fulfilled, with their objects replaced by whatever function of e they signify, in order for r to be granted. These could be e itself, all sources of e, the original version of e, and so forth. n may be zero, in which case L immediately grants r.

Figure 5 illustrates one step of the application of a license to obtain a desired concrete right r. In the license of r's object

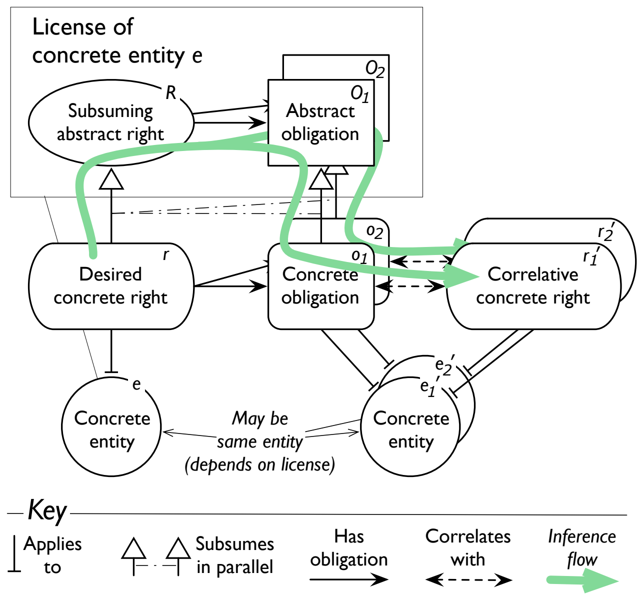


Fig. 5. A step in a rights/obligations inference

e, we search for an abstract right R subsuming r. The figure shows two obligations O_1 and O_2 of R, which we apply to r's object e in order to obtain r's concrete obligations o_1 and o_2. Depending on what kind of object O_1 has, o_1 could apply to e itself, in which case $e = e'_1$, or to an entity related to e, or (if L is a propagating license) to another module linked or otherwise connected to e. Finally, in order to fulfill o_1 we must have o_1's correlative right r'_1. The same considerations apply for O_2, of course. The heavy arrow shows the flow of inference from desired concrete right through to required concrete obligations and correlative rights.

If r'_1 (r'_2) is immediately available, its branch of the inference is complete. If not, the process recurses from r'_1 (r'_2).

The license rights and obligations for an entire system are calculated by repeating this process for every module of the system. If all modules are under the same license, analogous rights and obligations obtain for every module. If the system is heterogeneously-licensed, however, the calculation is much more varied, and if some of the modules are propagationally licensed then a right for one of those modules can produce obligations for other modules of the system. Such an architecture can easily result in license conflicts, as for example when a license propagates the obligation to be sublicensed under the same license to a proprietary component whose license forbids sublicensing. In such a case, the calculation will fail to produce a simultaneously satisfiable collection of obligations, and no rights will be available for the system as a whole.

Figure 6 shows in Toulmin form a portion of an example inference that produces a conflict, involving a component `e1` obtained under GPLv2 and modified, linked to a component `e2` obtained under the proprietary Corel Transactional License (CTL) [1]. The architectural connection between `e1` and `e2` is one that is interpreted for this inference as propagating

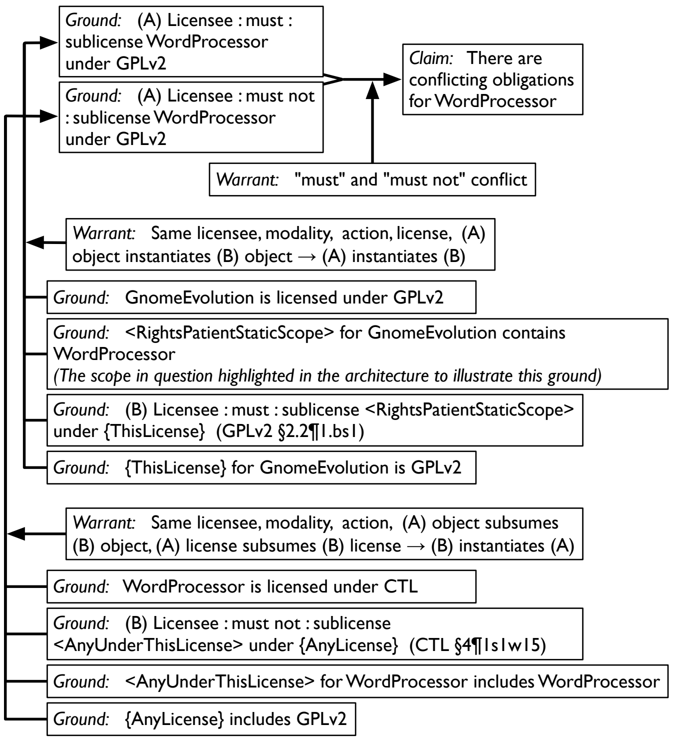

Ground: (A) Licensee : must : sublicense WordProcessor under GPLv2
Ground: (A) Licensee : must not : sublicense WordProcessor under GPLv2
Claim: There are conflicting obligations for WordProcessor
Warrant: "must" and "must not" conflict
Warrant: Same licensee, modality, action, license, (A) object instantiates (B) object → (A) instantiates (B)
Ground: GnomeEvolution is licensed under GPLv2
Ground: <RightsPatientStaticScope> for GnomeEvolution contains WordProcessor
(The scope in question highlighted in the architecture to illustrate this ground)
Ground: (B) Licensee : must : sublicense <RightsPatientStaticScope> under {ThisLicense} (GPLv2 §2.2¶1.bs1)
Ground: {ThisLicense} for GnomeEvolution is GPLv2
Warrant: Same licensee, modality, action, (A) object subsumes (B) object, (A) license subsumes (B) license → (B) instantiates (A)
Ground: WordProcessor is licensed under CTL
Ground: (B) Licensee : must not : sublicense <AnyUnderThisLicense> under {AnyLicense} (CTL §4¶1s1w15)
Ground: <AnyUnderThisLicense> for WordProcessor includes WordProcessor
Ground: {AnyLicense} includes GPLv2

Fig. 6. Toulmin-structured arguments supporting (and explaining) a typical conflict between obligations for a GPLv2 and a proprietary component

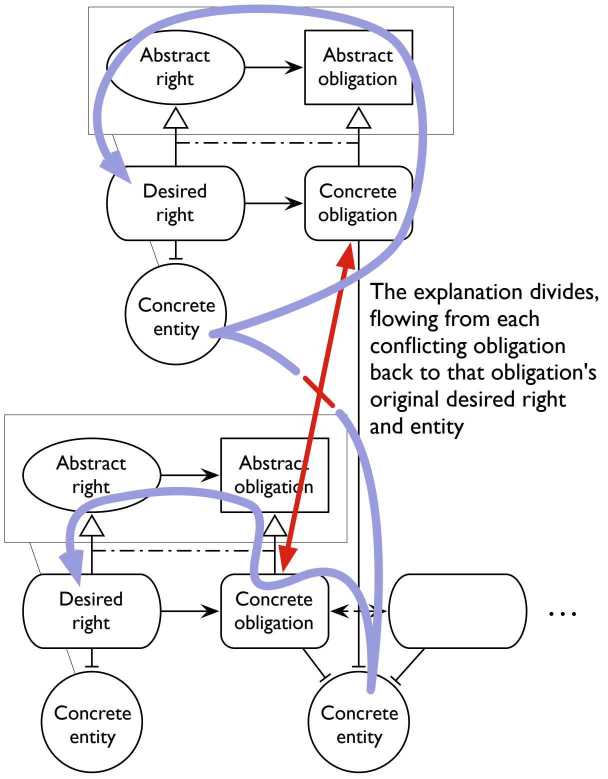


Fig. 7. Divided explanation flow for a conflict between two obligations

GPLv2 obligations, such as a static link. The right to distribute copies of the containing system is desired. In our prototype implementation (Figure 8) these arguments are presented in outline form, with the claim as the root of the outline and its grounds and warrant as its subheads, to be expanded as desired if further explanation is needed. A typical use would be:

1) Why does the WordProcessor component need to be sublicensed under GPLv2?
2) It is in the static-linked scope of the GnomeEvolution component; that component is annotated with the GPLv2 license; and GPLv2 obligates sublicensing under GPLv2 (GPLv2 §2.2¶1.bs1).
3) Why can't the WordProcessor component be sublicensed under GPLv2?
4) The WordProcessor component in the architecture has been annotated with the CTL license, and CTL forbids sublicensing under any license (CTL §4¶1s1w15).

B. Explanation by Argumentation

Figure 7 shows the two explanation flows for a conflict between obligations. Each flow begins at the conflict and explains how one half of the conflicting pair came to be. The connection between the pair is straightforward, as they are identical except for their modalities which are always "must" for one and "must not" for the other.

The flow and the required explanations are analogous for a right-obligation conflict, with the right and obligation again identical except for their modalities, which are always opposites, either "may" and "must not" or "must" and "need not".

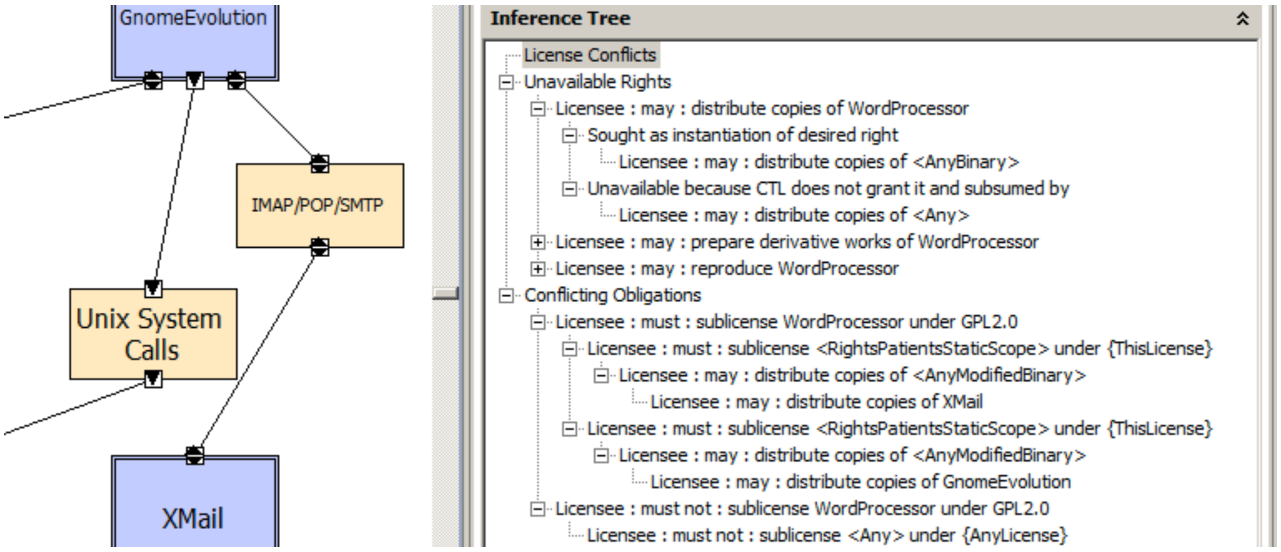
After examining the kinds of information that are available in the vicinity of a problem (a conflict or unavailable right), we realized the inferences leading up to it provide the clearest insight into what the problem signifies and why it is present.

- The chains of inference leading up to the problem constitute precisely the portion of the calculation relevant to the problem. No other parts of the calculation—or of the applications of license provisions, determined by the architecture and its annotations, that the calculation identifies—affect whether the problem is present or not.
- The inferences place the problem in the context of licenses, components and their annotations, and architectural configuration — the context in which a designer using the tool is already working.
- Each chain of inference, followed in reverse, provides an unfolding explanation for the problem's presence, which an analyst can explore as far as is helpful in providing understanding and insight.

Each step of a chain of inference is a point at which it can be broken—by replacing a component with one differently licensed, replacing one or more connectors to firewall off a propagating obligation, replacing a build-time component with one provided by users at run time, or other design decisions.

C. Automation

The license metamodel, calculation, and an assortment of license interpretations are implemented in a Java package. The



```
Inference Tree
 License Conflicts
 ⊟ Unavailable Rights
   ⊟ Licensee : may : distribute copies of WordProcessor
     ⊟ Sought as instantiation of desired right
         Licensee : may : distribute copies of <AnyBinary>
     ⊟ Unavailable because CTL does not grant it and subsumed by
         Licensee : may : distribute copies of <Any>
   ⊞ Licensee : may : prepare derivative works of WordProcessor
   ⊞ Licensee : may : reproduce WordProcessor
 ⊟ Conflicting Obligations
   ⊟ Licensee : must : sublicense WordProcessor under GPL2.0
     ⊟ Licensee : must : sublicense <RightsPatientsStaticScope> under {ThisLicense}
       ⊟ Licensee : may : distribute copies of <AnyModifiedBinary>
           Licensee : may : distribute copies of XMail
     ⊟ Licensee : must : sublicense <RightsPatientsStaticScope> under {ThisLicense}
       ⊟ Licensee : may : distribute copies of <AnyModifiedBinary>
           Licensee : may : distribute copies of GnomeEvolution
   ⊟ Licensee : must not : sublicense WordProcessor under GPL2.0
       Licensee : must not : sublicense <Any> under {AnyLicense}
```

Fig. 8. Prototype explanation results for a CTL-GPL2.0 conflict: (at top) unavailable rights (partially collapsed), (middle) two conflicting obligations.

calculation builds the entire dag, which is then available for presentation in whatever ways are desired. Each abstract right and obligation in a license interpretation has its provenance in the license or interpretation for use in explanations. The package supports the addition and use of new interpretations.

The package is connected into the system design context by its integration into an ArchStudio 4 plugin [4]. The plugin maps features of software architectures onto the license architecture abstraction needed for the virtual license calculation and displays results in the context of the architecture.

The argument grounds drawn from the texts of licenses are implemented through URLs hyperlinking into our collection of software licenses tagged for reference with §-¶-sentence-word numbers [1]. Each URL cites the sentence or phrase from which a right or obligation arises. Word-level `ids` allow references to, for example, #S2.2p1.bs1w11 for the phrase beginning at word 11 of that sentence.

VI. Conclusion

HtL system design and development provide important benefits but impose new demands difficult to meet using only manual methods and human insight. Our approach for supporting HtL development and acquisition automates the calculation of HtL system virtual licenses. We have integrated it into a software architecture tool so it can be applied at the point in the development process when the necessary information is available and the relevant design decisions are made. A key benefit it provides is the automated calculation of license conflicts, desired but unavailable rights, and virtual licenses. But explaining them is of even greater value.

We present a novel approach that presents each conflict in the form of structured arguments showing why each conflict exists and (by implication) points of attack for eliminating it. These arguments provide an informative presentation that brings together all the available information in a compact, evocative form that is easier to interpret, act on, and verify.

Acknowledgments

This research supported by grant #0808783 from the U.S. National Science Foundation, and grant #N00244-10-1-0077 from the Acquisition Research Program at the Naval Postgraduate School. No review, approval, or endorsement is implied.

The authors thank the anonymous reviewers of earlier versions of this paper for their insightful suggestions.

References

[1] T. A. Alspaugh. OSS (and other) licenses, §/¶/sentence/word-numbered. http://www.thomasalspaugh.org/pub/osl-sps/.

[2] T. A. Alspaugh, H. U. Asuncion, and W. Scacchi. Intellectual property rights requirements for heterogeneously-licensed systems. In *17th Int. Requirements Engineering Conference (RE'09)*, pages 24–33, 2009.

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

[4] E. Dashofy, H. Asuncion, S. Hendrickson, et al. Archstudio 4: An architecture-based meta-modeling environment. In *28th Int. Conference on Software Engineering, Companion Volume*, pages 67–68, 2007.

[5] L. Determann. Dangerous liasons—software combinations as derivative works? *Berkeley Technology Law Journal*, 21(4), 2006.

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

[7] C. B. Haley, R. Laney, J. D. Moffett, and B. Nuseibeh. Security requirements engineering: A framework for representation and analysis. *IEEE Transactions on Software Engineering*, 34(1):133–153, 2008.

[8] W. N. Hohfeld. Some fundamental legal conceptions as applied in judicial reasoning. *Yale Law Journal*, 23(1):16–59, 1913.

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

[10] P. Oreizy. *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution*. PhD thesis, Univ. of Calif., Irvine, 2000.

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

[12] M. J. Sergot, F. Sadri, et al. The British Nationality Act as a logic program. *Communications of the ACM*, 29(5):370–386, May 1986.

[13] M. L. Stoltz. The penguin paradox: How the scope of derivative works in copyright affects the effectiveness of the GNU GPL. *Boston University Law Review*, 85(5):1439–1477, 2005.

[14] S. Toulmin, R. Rieke, and A. Janik. *An introduction to reasoning*. Macmillan, 1984.