UCI-AM-12-195

ACQUISITION RESEARCH SPONSORED REPORT SERIES

Investigating Advances in the Acquisition of Secure Systems Based on Open Architectures

30 August 2012

A Compilation of Reports by

Thomas A. Alspaugh, Project Scientist

Walt Scacchi, Senior Research Scientist

Institute for Software Research

University of California, Irvine

With contributions from

Craig Brown, Programmer/Analyst

Kari Nies, Programmer/Analyst

Institute for Software Research

University of California, Irvine

Rihoko (Inoue) Kawai,

Associate Professor, Saitama Institute of Technology

Hazeline U. Asuncion, Assistant Professor

Computing and Software Systems

University of Washington, Bothell

Prepared for: Naval Postgraduate School, Monterey, California 93943

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

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

To request defense acquisition research, to become a research sponsor, or to print additional copies of reports, please contact any of the staff listed on the Acquisition Research Program website (www.acquisitionresearch.net).

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

About the Authors

Thomas A. Alspaugh is a project scientist at the Institute for Software Research, University of California, Irvine. His research interests are in software engineering, requirements, and licensing. Before completing his PhD, he worked as a software developer, team lead, and manager in industry, and as a computer scientist at the Naval Research Laboratory on the Software Cost Reduction, or A-7 project.

Thomas A. Alspaugh
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455
Tel: 949-824-4130
Fax: 949-824-1715
E-mail: alspaugh@ics.uci.edu

Walt Scacchi is a senior research scientist and research faculty member at the Institute for Software Research, University of California, Irvine. He received a PhD in information and computer science from UC Irvine in 1981. From 1981 to 1998, he was on the faculty at the University of Southern California. In 1999, he joined the Institute for Software Research at UC Irvine. He has published more than 150 research papers and has directed 60 externally funded research projects. In 2012, he serves as general co-chair of the Eighth IFIP International Conference on Open Source Systems (OSS2012).

Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455
Tel: 949-824-4130
Fax: 949-824-1715
E-mail: wscacchi@ics.uci.edu

Craig Brown was a programmer/analyst at the Institute for Software Research, University of California, Irvine, until Fall 2011. He received a B.S. In Information and Computer Science at UCI in 2009, and then joined ISR. Starting Fall 2011, he entered the M.S. Program in Game Design and Video Game Development at The Guildhall at Southern Methodist University, and expects to complete his degree in Spring 2013.

Craig Brown
The Guildhall
Southern Methodist University
Cbrown84@gmail.com

Kari Nies is a senior programmer/analyst at the Institute for Software Research, University of California, Irvine. She received a B.S. and M.S in Information and Computer Science from UCI. She has been at ISR since 1990.

Kari Nies
Institute for Software Research
University of California, Irvine

Irvine, CA 92697-3455
Tel: 949-824-8756
Fax: 949-824-1715
kari@ics.uci.edu

Rihoko (Inoue) Kawai, is an associate professor on the Faculty of Human and Social Studies, Department of Information Society Studies, at the Saitama Institute of Technology in Saitama, Japan**.**

Rihoko (Inoue) Kawai,
Saitama Institute of Technology
Saitama, Japan
rihoko@nii.ac.jp

Hazeline U. Asuncion, is an assistant professor in the Department of Computing and Software Systems at the University of Washington, Bothwell. Her research interests focus on software architecture, workflows, and software acquisition.

Hazeline U. Asuncion
Computing and Software Systems
University of Washington, Bothell
Bothell, WA 98011-8246 USA

Acknowledgments

Support for this research and the production of research publications included comes primarily from grant #N00244-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School. Additional support also contributing to this effort includes grant #N0024-10-1-0064 from the Center for Edge Power, also at the Naval Postgraduate School, and grant #0808783 from the National Science Foundation. No review, approval, or endorsement is implied.

THIS PAGE INTENTIONALLY LEFT BLANK

UCI-AM-12-195

ACQUISITION RESEARCH SPONSORED REPORT SERIES

Investigating Advances in the Acquisition of Secure Systems Based on Open Architectures

30 August 2012

A Compilation of Reports by

Thomas A. Alspaugh, Project Scientist
Walt Scacchi, Senior Research Scientist
Institute for Software Research
University of California, Irvine

With contributions from

Craig Brown, Programmer/Analyst
Kari Nies, Programmer/Analyst
Institute for Software Research
University of California, Irvine

Rihoko (Inoue) Kawai,
Associate Professor, Saitama Institute of Technology

Hazeline U. Asuncion, Assistant Professor
Computing and Software Systems
University of Washington, Bothell

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

Executive Summary

In 2007–2008, 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 2011–2012 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 investigate, discover, develop and document foundations for emerging policy and guidance for acquiring software systems that require OA and that incorporate OSS elements.

The research described in this final report for the 2011–2012 project year focuses on continuing investigation and refinement of techniques for reducing the acquisition costs of complex software systems. Over the past few years, we have investigated, demonstrated, and refined techniques and tools that articulate interaction between system requirements and software architecture that can increase or decrease the cost of software system acquisition. We have developed software architecture modeling techniques, notational schemes, and formal logic that can be incorporated into automated tools that allow for the construction of open architecture systems using proprietary and open source software components. Such capabilities allow for increased choice and flexibility for how best to satisfy system requirements through alternative software architectures that can accommodate different components and system configurations. Such capabilities help reduce system acquisition costs. In the proposed effort, we continue these investigations by incorporating reusable software product line (SPL) techniques within OA systems composed from proprietary and open source software (OSS) components subject to different intellectual property rights licenses, and where software components are subject to different security requirements. The combination of SPLs and OSS components within secure OA systems represents a significant opportunity for reducing the acquisition costs of software-intensive systems by the DoD and other government agencies.

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 secure software systems that rely on OA, OSS, and SPLs. In particular, it includes six research papers that have been refereed, reviewed, presented, and published in national and international research conferences, symposia, and workshops.

The first of these papers (Scacchi & Alspaugh, 2012) was originally presented at the 2012 Acquisition Research Symposium in May 2012. Five other refereed research

papers were also produced and are included in this report. In particular, one paper investigates how OSS and game-based virtual worlds were leveraged in a related study (Scacchi, Brown, & Nies, 2012) co-sponsored by the Center for the Edge Power at the Naval Postgraduate School, where we applied these methods to explore and demonstrate the ability to rapidly prototype SPL-based OA systems for military command and control (C2) systems. We believe this is a promising related line of research that merits further study.

Finally, to help contextualize our research effort and results, we provide some background on emerging issues in the acquisition of software-intensive systems that require OA and encourage or embrace the utilization of OSS, such as rapid, distributed evolution to meet immediate warfighter needs and its interplay with validation and system management. Last, we identify the results from our research efforts in the form of refereed publications. These publications cover the problems, issues, opportunities, and approach for acquisition research we identified for study during the 2011–2012 project effort.

Investigating Advances in the Acquisition of Secure Systems Based on Open Architectures

Overview and Background for this Research

Across the three military Services within the Department of Defense (DoD), *open architecture* (OA) and *OA systems* mean different things and are seen as the basis for realizing different kinds of technological, economic, and acquisition outcomes (Scacchi & Alspaugh, 2008). Thus, it is unclear whether the acquisition of a software system that realizes an OA, as well as one that utilizes open source software (OSS) technology and development processes, 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 the DoD. But there is now more explicit guidance for how best to develop, deploy, and sustain complex software-intensive military systems utilizing OA and OSS components (DoD Open Systems Architecture [DoDOSA], 2011; Hissam, Weinstock, & Bass, 2010).

Security is an essential issue in military software acquisition. However, we have found little effective guidance for addressing it in ways that can take advantage of the characteristics of OA systems based on software product lines (SPLs).Thus, there is an essential need for new knowledge and process guidance that program managers and others in the acquisition workforce can readily use to realize the potential benefits arising from SPL-based OA systems. Further, we anticipate that with growing awareness of emerging cyber warfare threats, the security of OA systems will potentially be mandated and thus become part of program acquisition processes. This in turn raises concern about potential cost growth and whether the acquisition workforce is well prepared to provide the needed oversight, review, and approval. The Software Assurance Acquisition Working Group's extensive report (Polydys & Wisseman, 2008) makes clear how important security is in software acquisition; it is mentioned on most pages of the report. The recommended approaches for improving the security of software systems involve manual reviews and process improvements. Reviews by experts are recommended and discussed for requirements, architectures, components, tests, and so forth, with a new review required for each new version; we believe this could constitute a serious and time-consuming burden, especially in the context of tight budgets, short timelines, and reductions in the acquisition workforce. However, no specific guidance is offered for OA or SPL systems by these reports. CMU's Software Engineering Institute is prominent in SPL research and practical guidance. The current Framework for Software Product Line Practice (Northrop and Clements, 2007) mentions security as one of the desired quality goals or attributes, along with reliability, usability, and others, but offers no specific guidance for addressing it. Recent papers from SEI presented at the Acquisition Research Symposium (Bergey & Jones, 2010; Jones & Bergey, 2011) discuss the benefits of an SPL

strategy but do not explain how security fits into the specification or documentation of OAs.

There appears to have been comparatively little research published on the topic of security and software product lines, even without considering open architecture, and very little on security and open architecture. One of the stronger examples (Mellado, Fernández-Medina, & Piattini, 2010) summarizes a line of research over four years that addresses security requirements for software product lines (but not OAs), and guides requirements activities with the goal of addressing security concerns in a way that corresponds to the characteristics of a software product line. Their approach is process-oriented, supported by a research tool that manages the various repositories of information that are developed, but informal and primarily manual and dependent on expert practitioners. Only the requirements phase of development is addressed.

The current guidance for program managers acquiring OA systems (DoDOSA, 2011) points to the need to identify and review the use of "security engines" that can support security enforcement tasks within system development or deployment. Similarly, current guidance on best practices on improving cost effectiveness in program acquisition (Better Buying Power [BBP], 2012) offers no clear directions for how best to address or manage specific cost issues that arise during secure OA system acquisition. However, recent acquisition research indicates there is also a need for a more articulate and streamlined process that acquisition workers can follow to insure that all relevant aspects of OA system security have been addressed in an easy to review format (cf. Scacchi & Alspaugh, 2012a). Similarly, current research further points to the need to address software reuse (Mactal & Spruill, 2012) and testing processes (Berzins, 2012) when SPLs are employed in OA systems as cost reduction and quality improvement strategies.

Most of the guidance to date for acquisition of secure open architecture software product lines may be summarized as follows: collect experts in security requirements, architecture, and tests; have them review and/or guide the manual/informal development of requirements, architecture, and tests for the product line; and repeat all or selected parts of the process for each new version and product line instance. We find little guidance for incorporating formality, for effectively using software tools, and then addressing and taking advantage of the specific characteristics of OA and SPL. Similarly, there is little guidance regarding the best process to identify and review OA system security when incorporating OSS components and other reusable SPL elements. There is no guidance for how to adapt or streamline such a process to reduce costs of acquiring different kinds of systems (e.g., enterprise information systems, command and control systems, embedded weapon systems), nor what information to consider or knowledge to acquire to enable a more effective acquisition workforce.

Consequently, this leads us to consider the following questions: What is the most effective way to articulate a process for the most cost-effective acquisition of secure OA systems that can be streamlined to the needs of specific kinds of reusable software systems? What issues or research questions for acquisition research follow from such a problem? What research approach can best explore the opportunities for acquisition research built from related research efforts in OA, reusable SPL, and software architectural analysis that can also inform future acquisition cost reduction practices

and improve acquisition workforce capabilities? We briefly elaborate on these questions, in turn, through the remainder of this proposal.

Problem for Acquisition Research

OAs imply software system architectures incorporating OSS components and proprietary components with open application program interfaces (APIs) that also conform to open standards (DoDOSA, 2011). Given the goal of how best to acquire *secure* OA systems, together with the use of evolving OSS components, reusable SPLs, and proprietary system components with open APIs, how should program acquisition and cost reduction processes, system requirements, software verification and validation (V&V), open architectures, and post-deployment system support be aligned to achieve this goal? This is a core research problem we seek to investigate. We seek to identify principles, best practices, and process activities for how best to ensure the success of the OA strategy when security requirements must be addressed with system components that may or may not be secure. Without such knowledge, program acquisition managers and Program Executive Offices (PEOs) are unlikely to acquire reusable 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 secure OA systems with OSS components and reusable SPLs (Scacchi & Alspaugh, 2011, 2012a), this research project also seeks to explore and answer the following kinds of research questions: How does the interaction of requirements and architectures for secure OA systems incorporating OSS components facilitate or inhibit acquisition processes over time? What are the best available processes for continuously verifying and validating the functionality, correctness, openness, and security of OA when OSS components and SPLs are employed? How can the use of continuously evolving OSS within a reusable OA or SPL be combined with the need to verify and validate critical systems security requirements and to manage their evolution? How do reliability and predictability trade-off against the cost and flexibility of a secure OA system when incorporating reusable SPL components? How should secure 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?

Inter-Project Research Coordination

We believe we are extremely well positioned to leverage our current research work and results (e.g., Alspaugh, Asuncion, & Scacchi, 2009a, 2009b; Scacchi & Alspaugh, 2008, 2011, 2012a, 2012b) with the effort describe in this report. We build on our current research efforts in OSS (Scacchi, 2007, 2010) and software requirements-architecture interactions (Scacchi, 2009; Scacchi & Alspaugh, 2008), as well as our track record in prior acquisition research studies. Similarly, we find current related research supported by the DoD addressing related issues in OSS (Hissam et al., 2010) also influences our

proposed effort. In addition, our effort builds from and contributes to research on software system acquisition within the DoD, focusing on software reuse (Mactal & Spruill, 2012), SPLs (Bergey & Jones, 2010; Guertin & Clements, 2010), open innovation and emerging software component markets (Guertin & Womble, 2012), efficient testing of component-based OA systems and SPLs (Berzins, 2012), and the improvement of software system acquisition through workforce upgrades and government–industry teaming (Heil, 2010). We thus believe our complementary research places us at an extraordinary advantage to conduct the proposed study that addresses a major strategic acquisition goal of the DoD and the military services (DoDOSA, 2011).

Prospects for Longer-Term Acquisition-Related Research

The military Services have 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 (DoDOSA, 2011). Thus, it would seem that 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 (including contract management and software development issues), and (b) how secure, reusable software product lines that employ an OA incorporating OSS products and production processes are essential to improving the effectiveness of future, software-intensive program acquisition efforts.

Statement of Research Status and Results

The proposed effort continues to build on a line of research that starts with our paper, Scacchi and Alspaugh (2008), and continues through the most recent research paper, Scacchi and Alspaugh (2012a). Nearly twenty externally reviewed and published research papers have resulted from this sustained research effort, which includes papers presented at the Acquisition Research Symposium every year since 2008. This final report for 2011–2012 highlights many of the ways that the new effort builds on the prior efforts in areas addressing OSS, OA, SPLs, software licenses, and secure software—most recently, strategies and methods for specifying the requirements for acquiring secure OA software systems that conform to SPLs. In particular, the newly proposed effort focuses on articulating and streamlining the process, and also on identifying cost reduction opportunities for acquiring secure OA software systems conforming to reusable SPLs, in ways that produce new knowledge for the acquisition workforce. Last, we are proud to note that each of the following chapters in the remainder of this final report have been published in international research journals, conferences, or workshops, or included as chapters in invited edited books, and all have gone through peer review. We believe our efforts thus continue to bring advances in acquisition research in our area of specialization to broader academic, government, and industrial research audiences.

The publications in which each of the following chapters in this report appear are listed as follows:

1. Scacchi, W., & Alspaugh, T. A. (May, 2012). Addressing the challenges in the acquisition of secure systems with open architectures. In *Proceedings of the*

Ninth Annual Acquisition Research Symposium. Retrieved from http://acquisitionresearch.net

2. Scacchi, W., Brown, C., & Nies, K. (2012, June). Exploring the potential of virtual worlds for decentralized command and control. In *Proceedings of the 17th International Command and Control Research and Technology Symposium (ICCRTS)*. Retrieved from http://dodccrp.org/events/17th_iccrts_2012/post_conference/papers/096.pdf
3. Alspaugh, T. A., Asuncion, H. A., & Scacchi, W. (2012, in press). Software licenses, open source components, and open architectures. In I. Mistrik, A. Tang, R. Bahsoon, & J. A. Stafford (Eds.), *Aligning enterprise, system, and software architectures*. Hershey, PA: Business Science Reference.
4. Alspaugh, T. A., Asuncion, H. A., & Scacchi, W. (2012, in press). The challenge of heterogeneously licensed systems in open architecture software ecosystems. In S. Jansen, S. Brinkkemper, & M. Cusumano (Eds.), *Software ecosystems*. Cheltenham, UK: Edward Elgar.
5. Alspaugh, T. A., Scacchi, W., & Kawai, R. (2012, September). Software licenses, coverage, and subsumption. In *Proceedings of the Fifth International Workshop on Requirements Engineering and Law*. Retrieved from http://www.ics.uci.edu/~wscacchi/Papers/New/Alspaugh-Scacchi-Kawai-RELAW12.pdf
6. Alspaugh, T. A., & Scacchi, W. (2012, September). Security licensing. In *Proceedings of the Fifth International Workshop on Requirements Engineering and Law*. Washington, DC: IEEE Computer Society.

Last, we propose to continue this line of research study and results through ongoing studies that investigate how best to support the acquisition of secure OA systems that incorporate both OSS and proprietary closed source software system elements in ways that enable simple yet dramatic streamlining improvements in software system acquisition processes performed by program managers and others in the acquisition workforce. We thus welcome any comments, questions, or suggestions for how our research studies or results might best be aligned with new program acquisitions, especially those focusing on the development and deployment of next-generation, software-intensive command and control systems, whether for conventional or cyber command missions and operations.

References

Alspaugh, T. A, Asuncion, H., & Scacchi, W. (2009a). Software licenses, open source components, and open architectures. In *Proceedings of the Sixth Annual Acquisition Research Symposium* (Vol. 1, pp. 258–275). Retrieved from http://www.acquisitionresearch.net

Alspaugh, T. A, Asuncion, H., & Scacchi, W. (2009b, September). Intellectual property rights requirements for heterogeneously licensed systems. In *Proceedings of the 17th International Conference on Requirements Engineering (RE '09)* (pp. 24–33). Los Alamitos, CA: IEEE.

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

Better Buying Power (BBP). (2012). Better buying power (public site). Retrieved from Defense Acquisition University website: https://acc.dau.mil/CommunityBrowser.aspx?id=432727&lang=en-US?

Bergey, J., & Jones, L. (2010). Exploring acquisition strategies for adopting a software product line approach. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 111–122). Retrieved from http://www.acquisitionresearch.net

Berzins, V. (2012). Certifying tools for test reduction in open architecture. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 185–194). Retrieved from http://www.acquisitionresearch.net

Department of Defense Open Systems Architecture (DoDOSA, December). (2011). *Department of Defense open systems architecture contract guidebook for program managers* (Vol. 0.1). Retrieved from https://acc.dau.mil/OSAGuidebook

Guertin, N., & Clements, P. (2010). Comparing acquisition strategies: Open architecture versus product lines. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 78–90). Retrieved from http://www.acquisitionresearch.net

Guertin, N., & Womble, B. (2012). Competition and the DoD marketplace. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 76–82). Retrieved from http://www.acquisitionresearch.net

Heil, J. (2010). Enabling software acquisition improvement: Government and industry software development team acquisition model. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 203–218). Retrieved from http://www.acquisitionresearch.net

Hissam, S., Weinstock, C. B., & Bass, L. (2010). On open and collaborative software development in the DoD. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 219–235). Retrieved from http://www.acquisitionresearch.net

Jones, L., & Bergey, J. (2011). An architecture-centric approach for acquiring software-reliant system. In *Proceedings of the Eighth Annual Acquisition Research Symposium*. Retrieved from http://www.acquisitionresearch.net

Mactal, R., & Spruill, N. (2012). A framework for reuse in the DoN. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 149–164). Retrieved from http://www.acquisitionresearch.net

Mellado, D., Fernández-Medina, E., & Piattini, M. (2010). Automated support for security requirements engineering in software product line domain engineering. *Information and Software Technology, 52*(10), 1094–1117.

Northrop, L., & Clements, P. (with Bachmann, F. et al.). (2007). *A framework for software product line practice, version 5.0*. Retrieved from http://www.sei.cmu.edu/productlines/frame_report/index.html

Polydys, M. L., & Wisseman, S. (2008). Software assurance in acquisition: Mitigating risks to the enterprise. Retrieved from https://buildsecurityin.us-cert.gov/swa/acqact.html

Scacchi, W. (2007). Free/open source software development: Recent research results and methods. In M. Zelkowitz (Ed.), *Advances in Computers* (Vol. 69, pp. 243–295), Elsevier, New York.

Scacchi, W. (2009). Understanding requirements for open source software. In K. Lyytinen, P. Loucopoulos, J. Mylopoulos, & W. Robinson (Eds.), *Design requirements engineering: A ten-year perspective* (LNBIP 14, pp. 467–494). Springer Verlag, New York.

Scacchi, W. (2010). The future of research in free/open source software development. In *Proceedings of the ACM Workshop on the Future of Software Engineering Research (FoSER)* (pp. 315–319). Santa Fe, NM, ACM, New York.

Scacchi, W., & Alspaugh, T. (2008). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (NPS-AM-08-036). Retrieved from http://www.acquisitionresearch.net

Scacchi, W., & Alspaugh, T. (2011). Advances in the acquisition of secure systems based on open architectures. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1, pp. 50–69). Retrieved from http://www.acquisitionresearch.net

Scacchi, W., & Alspaugh, T. (2012a). Addressing challenges in the acquisition of secure software systems with open architectures. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 165–184). Retrieved from http://www.acquisitionresearch.net

Scacchi, W., & Alspaugh, T. (2012b). Understanding the role of licenses and evolution in open architecture software ecosytems. *Journal of Systems and Software,* 85(7), 1479-1494, July 2012.

Womble, B., Schmidt, W., Arendt, M., & Fain, T. (2011). Delivering savings with open architecture and product lines. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1, pp. 7–31). Retrieved from http://www.acquisitionresearch.net

Yau, S. S., & Chen, Z. (2006). A framework for specifying and managing security requirements in collaborative systems. In *Proceedings of the Third International Conference on Autonomic and Trusted Computing (ATC 2006)* (pp. 500–510). Lecture Notes in Computer Science, 2006, vol. 4158, Springer, New York.

Addressing Challenges in the Acquisition of Secure Software Systems With Open Architectures

Walt Scacchi &Thomas A. Alspaugh

Abstract

We seek to articulate and address a number of emerging challenges in continuously assuring the security of open architecture (OA) software systems throughout the system acquisition life cycle. It is now clear that future systems must resist coordinated international attacks on vulnerable software-intensive systems that are of high value and control complex systems. But current approaches to system security are most often piecemeal with little or no support for guiding which system security requirements must be addressed across different system processing elements and data levels, and how those security requirements can be manifested during the design, building, and deployment of OA software systems. We present a framework that organizes OA system security elements and mechanisms in forms that can be aligned with different stages of acquisition—spanning system design, building, and run-time deployment, as well as system evolution. We provide a case study to show our scheme and how it can be applied to common enterprise systems.

Introduction

We seek to research, develop, and refine new concepts, techniques, and tools for continuously assuring the security of large-scale, open architecture (OA) software systems composed from software components that include proprietary/closed source software (CSS) and open source software (OSS). Federal government acquisition policy, as well as many leading enterprise information technology (IT) centers, now encourage the use of CSS and OSS, and thus OA, in the development, deployment, and evolution of complex, software-intensive systems.

We seek to prototype and demonstrate a new innovative approach and supporting technology that can develop new principles for correctness and security properties for OA systems. This includes developing basic principles to determine the security and performance properties of software systems, the conditions under which these properties hold, and the methods used to prove these properties of interest for systems. Of particular interest are networked OA software systems that are adapted or evolve to dynamic conditions and threats during their development, deployment, and usage, including those that may rely on new technologies like OA mobile devices (STIG 2012, Smalley, 2012) or other IT systems relying on open source technologies (CIO 2010; Garcia, 2010; Gizzi, 2011; Navy.mil, 2010). In particular, such study may be of value to securing new cyber warfare technologies (DoD 2011; Scacchi, Brown, & Nies, 2011). Our efforts may also lead to fundamental advancements for secure information sharing between information producers and consumers, in order to realize more secure information management, sharing, and interaction.

Challenges of Securing Systems With Open Architectures

Coordinated international attacks on vulnerable software-intensive systems that are of high value and control complex systems are becoming ever more apparent. As the StuxNet case demonstrates, security threats to software systems are multi-valent, multi-modal, and distributed across independently developed software system components ("Stuxnet," n.d.). Similarly, it is now clear that physically isolated/confined systems are vulnerable to external security attacks, via portable storage devices like USB drives, modified end-user devices (e.g., keyboards, mice ["Attack of the Computer Mouse," 2011]), and social engineering techniques (Sawers, 2011). This requires new security measures and policies necessary to defend such systems through new threat prevention and detection methods, as well as appropriate response mechanisms. Thus, what makes a system or system architecture secure changes over time, as new threats emerge and as systems evolve to meet new functional requirements. Consequently, there is need for an approach to continuously assure the security of complex, evolving OA systems in ways that are practical and scalable, yet robust, tractable, and adaptable. However, the best practices for developing OA systems whose components may be subject to differing security requirements (i.e., security rights and obligations) are unclear. Such practices are yet to be identified. This puts IT centers, system integrators, and service providers at a disadvantage when seeking to develop new software-intensive systems whose costs may be lower due to the integration of mature OSS components that are interfaced to pre-existing or new CSS components. OA systems thus present new challenges for assuring software system security.

Software system security mechanisms for enabling security requirements or policies are often employed on an ad hoc basis, since there are neither convenient or interactive tools, nor formal techniques, for specifying the security requirements of an OA system, or its components. Instead, what is available are disjoint mechanisms for implementing individual system security features (Loscocco et al., 1998; Spencer et al., 1999), such as the following:

- mandatory access control lists, firewalls;
- multi-level security;
- authentication (including certificate authority and passwords);
- cryptographic support (including public key certificates);
- encapsulation (including virtualization, hidden versus public APIs), hardware confinement (memory, storage, and external device [port] isolation; Sun, Wang, Zhang, & Stavrou, 2012), and type enforcement capabilities;
- secure programming practices (including secure coding standards, data type and value range checking; Seacord, 2008);
- data content or control signal flow logging/auditing;
- honey-pots and traps;

- security technical information guides for configuring the security parameters for applications (STIG 2012) and operating systems (Smalley, 2012);
- functionally equivalent but diverse multi-variant software executables (Franz, 2010; Salamat, Jackson, Wagner, Wimmer, & Franz, 2011).

But there is a gap between these mechanisms and any concept of a comprehensive security policy, whether for a system or any of its components, and no obvious way to integrate and evaluate them as a group. Similarly, it is unclear what relationships arise or are in place among these different security mechanisms. Further, what guidance is needed regarding which security mechanism to use where, when, why, and how, and how to update their usage or configuration as extant system security policy evolves? The mechanisms are also mostly software implementation choices rather than system architectural choices; no system-specific framework (like an architecture) exists in which software implementation choices can be pulled together in patterns that can be designed to meet specific security policies and goals. But in an OA system, it may be unclear or unlikely that system integrators will find mature OSS or CSS components that supply all of the system security features that the integrator or the customer requires on a timely, cost-effective basis.

Next, OA systems evolve through more pathways than traditional systems, as follows:

- individual components evolve through update revisions (e.g., security patches) made by the component's developers;
- individual components are updated with new, functionally enhanced versions from outside providers;
- individual components are replaced by different components from other sources;
- component interfaces evolve, either due to the system developers or outside sources;
- system architecture and configuration evolve as the developers adapt it to address new functional requirements;
- system functional and security requirements evolve, either due to the system developers, recognized gaps, or outside stakeholders.
- system security policies, mechanisms, security components, and system configuration parameter settings also change over time.

These additional evolution paths are tied to the benefits of using OA systems with OSS components but they also present new challenges for security. OA systems are continually evolving, and in our view, this fact is fundamentally unaddressed by prior work in security.

Beyond these issues, we must consider, how should customers specify what security system features they want their delivered systems to support? How can the history of security failures (vulnerabilities), faults (exploits), possible cyber-warfare attacks

(threats), and possible responses (updating system configuration with new elements that resist new threats, close new vulnerability, and prevent newly discovered exploits), to guide the evolution of approaches for developing secure OA systems? How can answers to questions like these help formulate a technological innovation element of the DoD strategy for operating in cyberspace (DoD 2011)? Questions like this remain unresolved at present.

Verification of the usage of security mechanisms in software systems is unclear, and often focused either at the whole system (macro) level, or program function or coding (micro) level, but generally not at the architectural component and interconnection (meso) level, and not for combinations and alternative configurations of CSS and OSS components with different security histories. We believe there is a new or under-explored opportunity to address security requirements at the architectural level. As such, we see the following basic challenges in assuring OA system security:

- how to verify the security of OA system designs throughout system development, deployment, and post-deployment support; and
- how to validate the effectiveness of OA system security measures, and feed back evolving knowledge of vulnerabilities and exploits into the ongoing development (continuous evolution) stream for existing and planned systems in an operational, testable form that system designers can use, and program managers can assess.

Similarly, we see the following basic challenges in assuring security of OA software systems:

- how best to develop complex OA systems whose OSS or CSS system components may originally come from trusted sources, but in which these components, the architectural configuration, and security requirements are subject to multiple sources of adaptation and evolution;
- how to go beyond "many eyes" (large number of skilled reviewers) to establish a scalable basis for automated or semi-automated verification of software system security properties as the system continually evolves;
- how to best achieve continuous software system security assurance as a system is adapted and evolved to address new security requirements and technology progress;
- how best to protect OA systems through biologically inspired natural defenses that provide adaptive and resilient mechanisms including agile response, isolation, and fail-soft recovery to immediate attacks, as well as adaptation via dynamic reconfiguration, multi-version mechanisms, (artificial) ecological diversity responses to sustained vulnerabilities or threats (Shrobe, 2011); and
- how to create reference models and security policy requirements that articulate security scenarios appropriate for oversight during system acquisition, as well as during system design, implementation, deployment, and beyond?

Securing Software Systems

The key ideas in our approach to develop and demonstrate a new solution to the challenges is to specify verifiable security requirements of OA systems using formalized "security licenses" (Scacchi & Alspaugh, 2011), and to use an explicit, evolvable software architecture to mediate and carry the paths of interactions among them. Security licenses must specify the security requirements and access/update rights and obligations within an OA system, its CSS and OSS components, and their interconnections (e.g., APIs, databases, shared files, communication protocols) that defend against threats and enable appropriate responses to attacks or suspicious/anomalous system behaviors. Subsequently, the goal of our approach is to articulate and refine the ways and means for expressing and verifying that the security requirements of OA system components match up appropriately and together support the security requirements of the entire OA system, at architectural design time, while enabling the automated verification of system builds/compositions and deployable, as well as of executable, run-time versions of the system.

Software licenses represent a collection of rights and obligations for what can or cannot be done with a licensed software component. Licenses can thus denote both functional and non-functional requirements that apply to software systems or system components during their development and deployment. But rights and obligations are not limited to concerns or constraints applicable only to software as IP. Instead, they can be written in ways that stipulate functional or non-functional requirements of different kinds. Consider, for example, that desired or necessary software system security properties can also be expressed as rights and obligations addressing system confidentiality, integrity, accountability, system availability, and assurance. This kind of approach provides new principles of correctness for software IP requirements (cf. Breaux & Antón, 2005, 2008).

Traditionally, developing robust specifications for non-functional software system security properties in natural language often produces specifications that are ambiguous, misleading, and inconsistent across system components, and lacking sufficient details (Yau & Chen, 2006). Using a semantic model and logic to formally specify the rights and obligations required for a software system or component to be secure (Breaux & Antón, 2005, 2008; Yau & Chen, 2006) means that it may be possible to develop both a "security architecture" notation and model specification that associates given security rights and obligations across a software system, or system of systems. Similarly, it suggests the possibility of developing computational tools or interactive architecture development environments that can be used to specify, model, and analyze a software system's security architecture at different times in its development—design time, build time, and run time. We have already demonstrated how such an approach can work, when limiting attention to IP rights and obligations. The approach we have been developing for the past few years for modeling and analyzing software system IP license architectures for OA systems (Alspaugh, Asuncion, & Scacchi, 2009; Alspaugh, Scacchi, & Asuncion, 2010; Scacchi & Alspaugh, 2008) , may therefore be extendable to also address OA systems with heterogeneous software

security license rights and obligations (Scacchi & Alspaugh, 2011). Furthermore, the idea of common or reusable software security licenses may be analogous to the reusable security requirements templates proposed by Firesmith (2004) at the Software Engineering Institute. Such security requirement templates may simplify and guide the efforts of customers (or contracting officers) to more readily specify workable requirements that can be readily verified through system development, deployment, and post-deployment support.

Security licenses can be specified, modeled, and analyzed continuously from initial system architectural design through post-deployment support and system evolution, with key points for security license analysis occurring at design time, build/linking time, and deployment/run time. Such security licenses can be stated both (a) informally, using restricted natural language for human readability, authorship, description of non-functional security requirements, as well as (b) formally, specifying functional security requirements in a computer-processable form using a logic-based scheme and modeling notation, with automated production of (a) from (b) and automated architecture-mediated inferences using (b). Analysis of system/s security requirements can therefore be integrated into the software architecture tool used to express and evolve the architecture, so that the analysis evolves automatically in parallel with the architecture.

In general terms, a security license is analogous to a software copyright license such as GPL (GNU General Public License; Free Software License, 2007). Software licenses consist of intellectual property (IP) *rights* granted by the license, and corresponding license *obligations* needed to obtain the rights. Our innovation is to similarly specify the security obligations and rights of OA system components using elements found in known security capabilities, which we can then model, analyze, and support throughout the system's development and evolution, and use to guide system design and instantiation. Our initial investigation of security licenses (Scacchi & Alspaugh, 2011) has identified rights and obligations such as the following:

- the obligation for a user to verify his/her authority to see compartment T, by password or other specified authentication process;
- the obligation for a specific component to have been vetted for the capability to read and update data in compartment T;
- the obligation for all components connected to specified component C to grant it the capability to read and update data in compartment T;
- the obligation to reconfigure a system in response to detected threats, when given the right to select and include different component versions, or executable component variants;
- the right to read and update data in compartment T using the licensed component;
- the right to replace specified component C with some other component;

- the right to add or update specified component D in a specified configuration;
- the right to add, update, or remove a security mechanism; and
- the right to update security license L.

Further, formally specified OA security licenses are verifiable, as well as grounded in functional and testable system security capabilities.

The security reasoning chains among the security licenses are mediated by the system architecture, and evolve automatically with it, much like they can for IP licenses (Alspaugh et al., 2009; Alspaugh, Asuncion, & Scacchi, 2011; Alspaugh et al., 2010). Each kind of security license details how its obligations are propagated architecturally to other system components. The results of this propagation, coupled with automated identification of gaps, conflicts, and subsumptions, are communicated to analysts as architecturally organized arguments supporting the existence of the identified issues. The arguments provide context-appropriate guidance, in terms of the system architecture and the security licenses of the components involved, for resolution of security problems through the evolution of the system design.

Our approach neither assumes nor proves that individual elements of an OA system are secure, but instead seeks to determine what security rights and obligations are in effect at any time for the overall system architecture as a function of the security rights and obligations of its components. This means that it is possible to configure a secure OA system whose components may be insecure, or not equally secure. Our approach also supports determination of where or how OA system security rights or obligations may be in conflict, mismatch, or subsume one another as individual system components or connectors are adapted to evolve over time. As an organization's security policies (i.e., their security requirements) evolve and adapt, the OA system's security rights and obligations are evolved to match and satisfy them, as long as all security requirements can be expressed through description logic relationships among them.

Security rights and obligations are characterized in terms of enterprise security policies and goals; within that closed world, our approach enables specification of the security properties that an open system architecture must match or satisfy. These security requirements also direct acquisition program managers' and architecture analysts' attention to problem areas with the greatest impact on system security. Where our approach identifies a conflict or mismatch, it indicates an actual, open-world weakness in the security of the OA system under analysis. The chain of reasoning is mediated by the system's architecture, with its units defined piecewise in each component's security license and evolving continuously as the system architecture, configuration, and security requirements evolve. As new kinds or types of vulnerability, threats, or exploits emerge, as well as new categories of effective responses and emerging alternative security mechanisms, we seek to elaborate and demonstrate how this approach can continuously accommodate the specification and analysis of changing security requirements.

Product Lines: Alternatives, Versions, Variants of OA Elements

In producing a secure OA system in a software product line, there are several levels of variation available for producing artificial diversity among equivalent instances and for selecting and evolving in the face of threats.

At the highest level of granularity, a system developer or integrator can choose among alternative *producers* of similar components, services, and platforms (Sun et al., 2012): For example, we can find *functionally similar* alternatives from software (component) producers of Web browsers like Mozilla (Firefox, Camino, Sea Monkey) versus Google (Chrome) versus Microsoft (Internet Explorer), versus others. Similarly, for word processors, we find alternatives including Microsoft (Word) versus abisoft.com (AbiWord) versus Google (Google Docs, which is a remote Web service rather than a component), versus others. Likewise, for email and calendar applications, we find alternatives like Microsoft Outlook, Gnome Evolution, Google Mail, and Google Calendar, among others. For operating systems, we find Red Hat Enterprise Linux, Microsoft Windows, Apple OSX, and Google Android, among others. Finally, note that some producers produce more than one alternative of the same kind of component or service, such as Mozilla's Web browsers (Firefox, Camino, SeaMonkey), so that a choice among those particular components does not result in a change of producers.

Functionally similar components and services may not be exactly interchangeable, unless their interfaces are similar or identical. As such, it may be necessary, for example, to modify OA system topology, or to replace connector types and other architectural measures may be necessary to change from one producer to another, depending on the functionality needed to satisfy functional requirements. However, in general, the overall functionality provided by the system remains substantially the same, but now the diversity among alternative system instances is the greatest: not only is the component, service, or platform distinct between two instances, but its architectural connections in the system will be distinct, as will be the software development process and organization that produced it, so the chances of a common vulnerability are greatly minimized. Subsequently, when functionally similar components, connectors, or configurations exist, such that equivalent alternatives, versions, or variants may be substituted for one another, then we have a strong relationship among these OA system elements that is called a *product family* (Narayanaswamy & Scacchi, 1987; Bosch, 2006) or a *product line* (Clements & Northrop, 2001).

As described above, a shift from one alternative to another ordinarily requires a change in architecture, software connectors, and other measures. Changes between some alternatives will also produce a change of producers, while others will not. However, when components or connectors provide alternative implementations of the functionality they provide, then these are designated as versions. For example, most Linux operating systems support multiple file systems for data storage, though developers or integrators select their preferred file system for inclusion at either design time or build time. Similarly, for connectors to remote Web servers, developers or

integrators may specify unencrypted (e.g., HTTP) or encrypted (e.g., HTTPS) data communication protocols for use in a Web-based enterprise system. Next, at the OA system configuration level, selection of alternative components or connectors, or of different versions of components or connectors result in different overall system versions that conform to a system product line. Further, recent advances in source code compilation now allow for creation of *functionally identical* variants of software components, though each variant has a different run-time image in the computer, through code randomization techniques (Franz, 2010; Salamat et al., 2011). Last, software product lines can be bound to a network of software producers, system integrators, and system users/consumers through a software ecosystem (Bosch, 2009), such that secure systems can be realized through composition or configuration at the software ecosystem level (Scacchi and Alspaugh 2012). Consequently, we now have a complete and robust basis for specifying OA systems that can include components, connectors, or application systems from alternative producers, or with different versions or variants included. This is now our basis for moving forward to address the challenges of creating secure OA systems through secured software product lines.

Secure Software Product Lines within an OA Software Ecosystem

Given the basis for software product lines for OA systems, we now address how to frame and align software system architectures with software security mechanisms. We use the following scheme to address this, as shown in Table 1.

Table 1. Different System Security Elements Whose Rights and Obligations Depend on Capabilities Supported by Lower Level Elements

Security policies	
Developers, system integrators and users	Persistent data
System configurations	
Components	Ephemeral data
Connectors	User I/O data
Platforms	

System security policies provide the overall context for the kinds of security mechanisms or capabilities (e.g., mandatory role-based data access control) required by a particular system. The requirements must be realized through multiple levels of system composition that span a processing space from people to processing platforms, and through data/content space that is processed during system usage/operation. Aligning system security elements with security mechanisms gives rise to the following associations:

Platform: base technological elements that constitute the computer environment that hosts the target system.

- Hardware: specifies hardware confinement constraints needed to securely operate the software system configuration, potentially to address memory, storage, and external device port isolation (see SecureSwitch [Sun et al., 2012]). Hardware may be configured as an embedded processor, mobile computer (e.g., smartphone or tablet), personal computer, multi-processor computation server, or multi-server data center.
- Virtual machine: a software layer that can isolate and confine the operating system, component applications, or application services from direct control of system hardware, network operations, or operating system processes. OSes, software systems, components, or connectors can each run within their own virtual machine, in alternative configurations, as long as they are completely confined at a higher level of system security and do not overlap virtual machine boundaries (Spencer et al., 1999; Smalley, 2012).
- Network: message filtering and access control firewalls for data/control flows that move across external hardware system security boundaries.
- Operating systems: mandatory access control (Loscocco et al, 1998; Spencer et al., 1999), capability type enforcement (Smalley, 2012), OS configuration parameters (STIG 2012), run-time audit logs, all currently coded and managed by system integrators/administrators.

Connectors: software mechanisms that implement secure communication mechanisms within and across system boundaries. Connectors enable security mechanisms providing:

- data cryptography (encryption/decryption) before/after data transfer;
- component-connector–specific firewalls that can be implemented via (pre-conditions) constraints on in-bound data flow and plug-in/helper application invocation, or on out-bound data flow and external program invocations (post-conditions); and
- multi-version connector configurations between components that allow for artificial diversity and dynamic reconfiguration potential through functionally similar versions.

Components: software mechanisms that implement application functionality required for the targeted system to operate as intended. Components enable security mechanisms providing:

- access/usage authentication control obligations (e.g., login with authorized identification and password) for which people in what roles (e.g., developer, system integrator, system administrator, system user) have the specified set of rights to view/update data, data control flow invocations, or external program invocations;

- encapsulated components as services within virtual machines to confine potential exploits, while mitigating their propagation;
- alternative versions that increase artificial diversity and enable dynamic replacement with functionally similar alternatives;
- multiple versions that allow for changes in vulnerability space, including concurrent versions with replicated input data, but different out data connector (routing) configurations; and
- multiple variants that reduce vulnerability to component version attacks.

System configuration: the composition and interrelationship of components and connectors that together realize the system architecture, at design time, build time, or run time. System configuration (or composition [Bosch, 2006]) enables security by providing the

- ability to host multiple (one or more) alternative, version, or variant system configurations on one or more processors (either single-core [Sun et al., 2012], multi-core, multi-blade, or multi-site) that can be dynamically selected in response to security policy directives or in response to detected threats;
- ability to host concurrently running multiple (one or more) alternative, version, or variant system configurations on one or more processors (either multi-core, multi-blade, or multi-site) that can be dynamically selected in response to security policy directives or in response to detected threats; and
- ability to (formally) specify system configuration as an open architecture at design time, build time, and deployment run time, along with automated tools that can verify the consistency, completeness, and traceability.

Developers, system integrators, and users: denote the people authorized and trusted to work on or with the configured systems or its elements over time, depending on their externally assigned role(s).

- Developers should employ software development environments, tools, or processes that reinforce security-safe software coding practices of components or connectors they implement as products (Seacord, 2008).
- Developers should produce multiple, unique executable variants of the components or connectors they produce and distribute.
- System integrators design OA system architecture.
- System integrators build OA system configurations that select from one or more component or connector alternatives, versions, and variants.
- System integrators deploy one or more run-time system configuration variants that can be readily installed and appropriate parameters entered by system administrators or end-users.
- System integrators or system administrators, or automated mechanisms under their control, must be able to monitor and access system execution

audit logs, to determine if threats or anomalous system behaviors are detected, and to dynamically reconfigure system configuration or security parameters in order to move the executable system into a more trusted operational state.

- Users must be provided with online identifiers or identification methods that enable them to access security controlled systems via one or more alternative authentication mechanisms in place.

In parallel with these processing security spaces are the following data security spaces:

User I/O data: data that may exist only as it passes across communication channels. Examples are keystrokes and mouse movements communicated from a keyboard or mouse to a processor, voice data from microphones and to speakers, wifi packets, and so forth. This data may be discarded or incorporated into ephemeral data.

Ephemeral data: data that exists in memory for a brief time before being either discarded or incorporated into persistent data. Examples are Web forms that have been filled out but not submitted, and data in various sorts of hardware buffers.

Persistent data: data that exists for a substantial time on local disks or solid-state storage devices, USB memory sticks, DVD-ROM, or server storage.

Security policies: provide overall guidance and requirements for what security mechanisms and regimes are to be designed, implemented, and satisfied during the deployment, operation, and evolution of a specified system. Security policies

- should provide non-functional requirements regarding the membership, structure, and behavioral specifications of each of the proceeding categories of security elements at minimum, or further specification of security sub-elements within each category, as per the security exposure of the system being addressed.
    - Non-functional requirements may only specify rights provided when corresponding obligations are fulfilled that cannot be automated or verified in lower level security elements.
    - Non-functional requirements should be expressible in human-readable and computer-processable forms within the system security policy license.
- must provide functional requirements regarding the membership, structure, and behavioral specifications of each of the proceeding categories of security elements at minimum, or further specification of security sub-elements within each category, as per the security exposure of the system being addressed.
    - Functional requirements are those that can be formalized, automated, and verified by corresponding automated mechanisms available at lower level security elements.

- Functional requirements may only specify rights provided when corresponding obligations are fulfilled that must be automated or verified in lower level security elements.
- Functional requirements should be expressible in human-readable and computer-processable forms within the system security policy license.

The case study that follows describes where these different system security elements appear in forms that can be available for review by authorized Program Acquisition personnel.

Case Study: A Secure Product Line for an Enterprise System

Let us consider what needs to be specified during the acquisition of an enterprise system that incorporates common office productivity applications that run on a personal computer networked to remote servers. Such a system can include a Web browser, word processor, email, and calendaring applications that are configured to operate on a personal computer, where the PC's operating system, Web browser, and other applications need to be configured to access remote data/Web content servers. Figure 1 shows part of the system ecosystem of software producers and the components they can provide for our enterprise system.

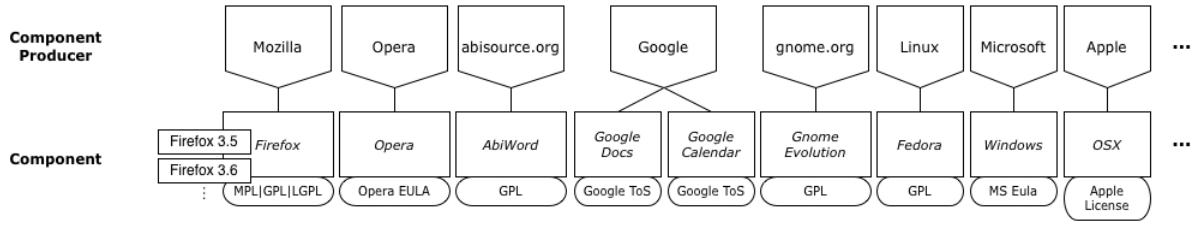

Figure 1. A Partial View of a Software Ecosystem of Producers and the Software Components for an Enterprise System They Produce

Figure 2 shows the design-time architecture of such an enterprise system. What might a secure product line for a system like this involve, and how might it provide benefits and security qualities to be specified for design time, build time, and run time? How can its OA and product-line characteristics contribute to security throughout the acquisition system life cycle?

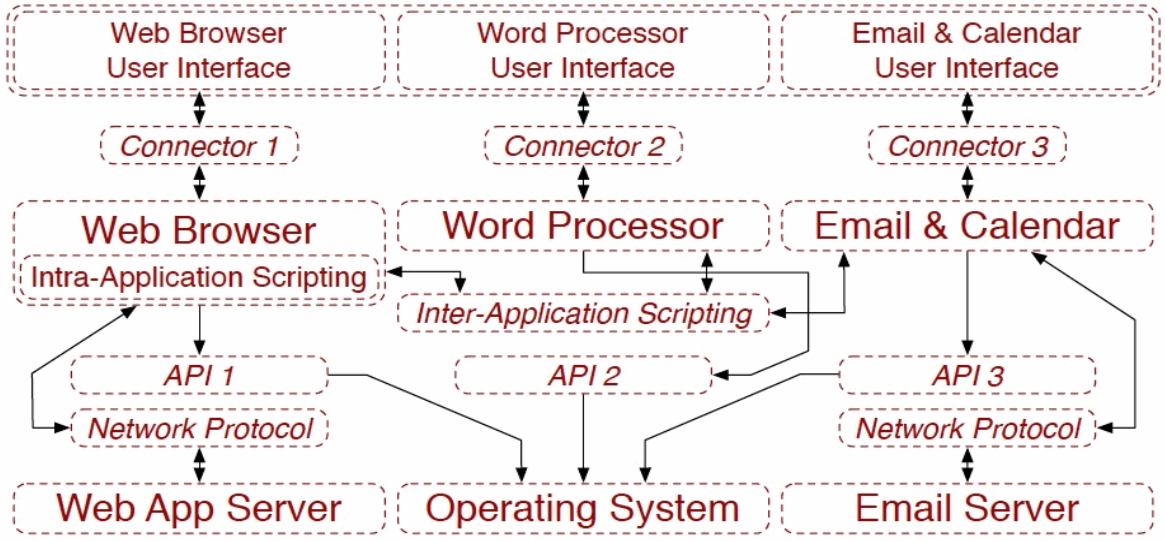

Figure 2. A Design-Time Reference Model of an OA System That Accommodates Multiple Alternative System Configurations

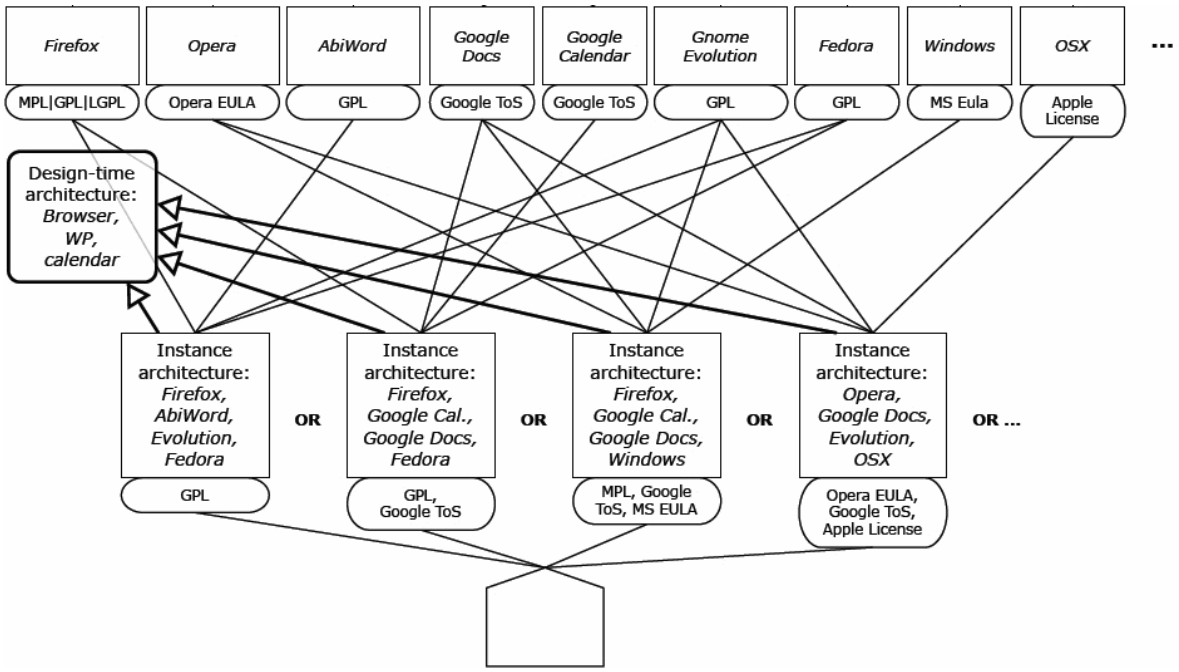

Figure 3. A View of an OA Software Ecosystem That Provides Alternative, Functionally Similar Components Compatible With the Reference Design-Time Architecture

We envision an approach in which non-functional requirements, such as security, reliability, and evolvability requirements at acquisition time, are elaborated at design and build times by specific functional requirements that explain how and to what degree the non-functional requirements are going to be satisfied at run time. Analogous to our previous work with intellectual property (IP) licensing, we envision that these requirements are structured in the same logical forms as IP licenses (with specific rights that are obtained only by fulfilling specific obligations), and managed through the

architecture by the same approach of calculating which obligations are satisfiable, in what way, and as a result what rights are available (Alspaugh et al., 2009; Alspaugh et al., 2010; Scachhi & Alspaugh, 2011).

Figure 3 illustrates a possible OA software ecosystem for this product line. Here a number of possible producers and alternative components have been placed into play, and four specific instance architectures (produced in four specific ecosystems) have been sketched. With appropriate architectural topologies, and appropriate shim components and connectors inserted between the major components, each of these four instance architectures can support the same functionality. It is also possible to achieve different nonfunctional qualities including security qualities through the four choices, for example, by requiring that OS be an appropriate Security-Enhanced version of Linux (SEL 2012), or by requiring that the network protocol connector be HTTPS.

Within the overall ecosystem of Figure 3, Figure 4 shows one possible instance ecosystem involving specific producers (Mozilla, abisource.org, gnome.org, Red Hat) and specific alternatives (Firefox, AbiWord, Evolution, Fedora).

Acquisition-time requirements such as the use of SE Linux and the use of HTTPS could be satisfied by this choice; with an appropriate architecture, the IP licensing obligations could also be satisfied. At design time, the functional requirements would need to be satisfied by appropriately specified shims inserted among the principal components, and if such shims could be designed, then this would be the proof that the acquisition-time nonfunctional requirements could also be satisfied. Figure 5 shows a run-time view of this instance architecture, resulting from the specific OA ecosystem and instantiating the overall ecosystem of Figure 3 and the software product line the software system is an instance of.

This instance architecture has both a manageable IP license regime that insures its openness, and a manageable security regime. For IP, in this architectural instance, all component versions can be selected to use permissive licenses (Web browser, Web server) or reciprocal GPL licenses (word processor, email, calendar, and operating system); they are cleanly separated by dynamic run-time links, which are a type of connector that does not transmit IP obligations or rights, though it allows for control flow integration, and data flow interoperation.

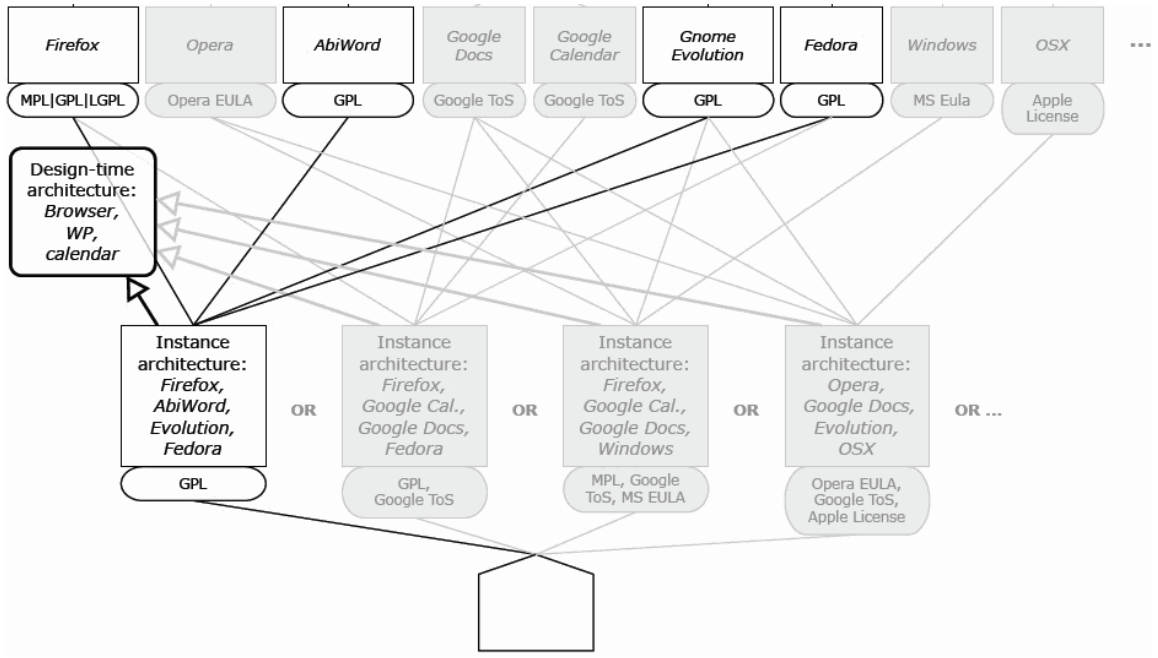

Figure 4. A Selection Among Alternative Components That Can Be Included at Build Time to Produce an Integrated System Compatible With the Design-Time Reference

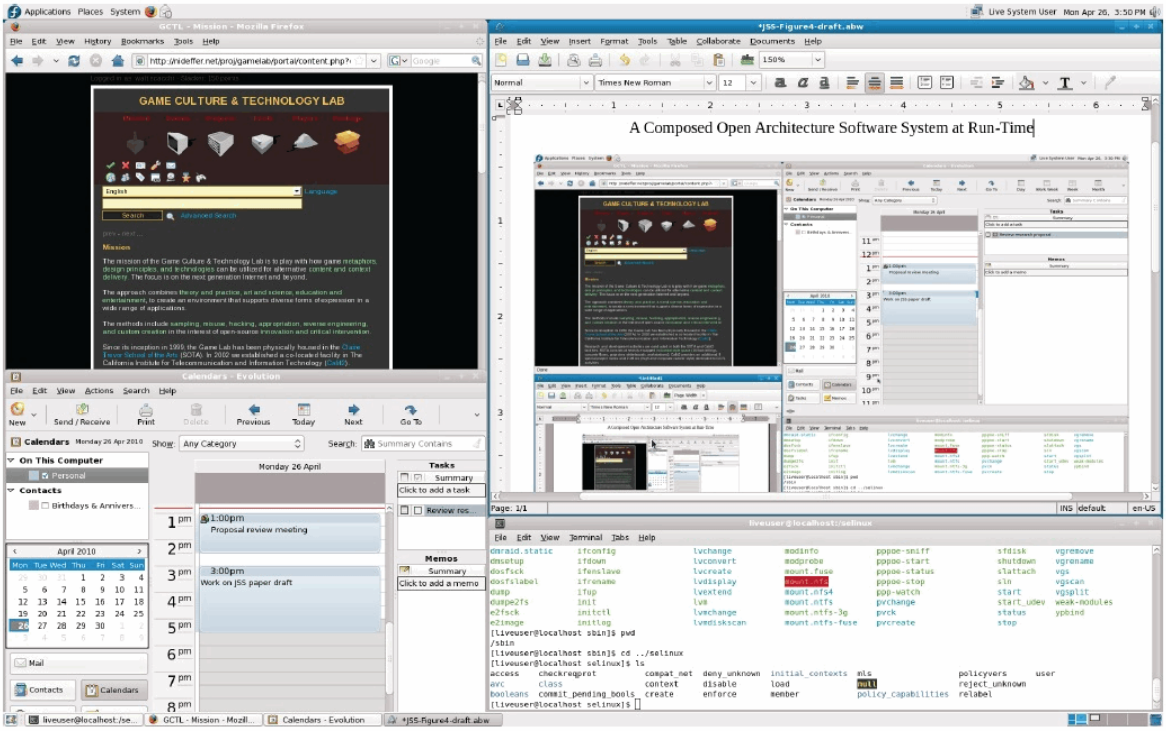

Figure 5. An End-User Run-Time Version of the Selected Alternative Components, utilizing Security-Enhanced Linux (SEL 2012).

Figure 6 outlines an alternative system configuration and the instance ecosystem that produces it. This instance architecture substitutes services for components in the case

of Google Docs for the word processing functionality and Google Calendar for the calendar functionality. With appropriate shims and changes to the architectural topology, this combination of major components could also support the system's functional requirements, and because the services are accessed through client–server connections, which block the propagation of most license obligations, there are a number of ways to satisfy the IP constraints imposed by the component and service licenses.

This alternative configuration also highlights possible acquisition-time concerns and the nonfunctional requirements and security license issues that follow from them. For example, a remote service such as Google Docs provides benefits and imposes costs with respect to a compiled component such as AbiWord. On the one hand, the remote service makes some qualities easier to achieve (data sharing, backup, etc.), but on the other hand, may make some qualities harder to achieve (data security over a network connection and in the "cloud," up-time of the service, little or no control over when new versions of the service are used, compared to complete control over when new versions of a component are integrated).

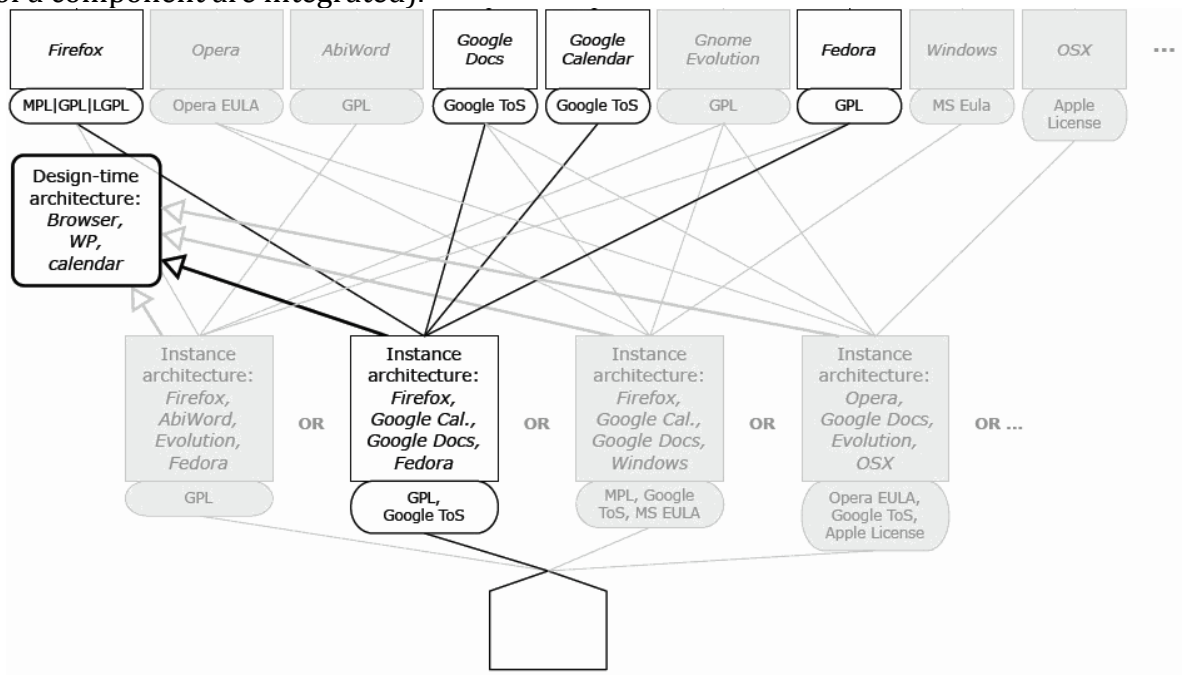

Figure 6. A Second System Configuration, Using Alternative but Functionally Similar Components

The alternative configuration gives rise to a number of questions addressing both acquisition-time concerns and for non-functional system requirements, such as:

- Who in the ecosystem of human actors for this system has the right to make the decisions to use a service in place of a component, or one component version in place of another? What obligations are they required to satisfy first? These questions are of concern at acquisition time and, we claim, are addressable by acquisition licenses that restrict rights and impose

obligations important to system acquisition officers just as IP licenses do for IP rights and obligations important to software producers.

- When can these decisions be made? In traditional development processes, these would occur at design time, but in the larger view we propound here, such decisions, or rather the policies or acquisition licenses that control them, are perhaps more properly considered at acquisition time. As explained in the remainder of this section, it is also possible that in order to achieve specific security qualities, they might be made at build or run time, in response to specific threats.

Figure 7 shows a run-time view of this alternative configuration. To the end user, this system appears quite similar to the one in Figure 5, and the differences might scarcely be noticed, which raises the next set of possibilities.

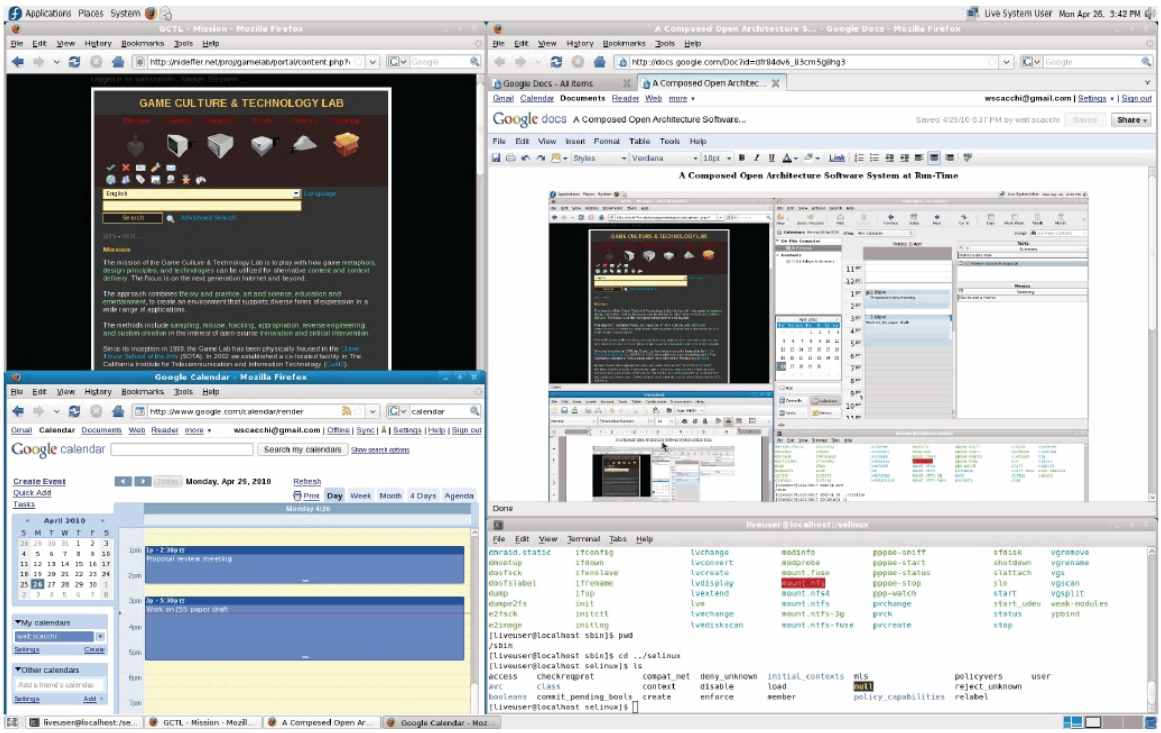

Figure 7. An End-User View of the Alternative Run-Time System Configuration

Both these instance architectures specify specific alternatives for the major components, for example, Firefox for the Web browser component. But which version of Firefox? For example, it is quite possible that both the instance architectures discussed above could be implemented using either Firefox 10 or Firefox 11, satisfying all the functional requirements with no change to the instance architecture and no revision of software shims. Who has the power to decide to use version 10 rather than version 11? How late in the software process can this decision be made—for example, could it be made as late as system startup time by a system user, in response to a particular security attack on the previous configuration?

At the conceptually lowest level, the advent of code randomization and multi-variant software executables leads to the possibility of substituting essentially equivalent variants of the same component, most obviously at build time. The decision to substitute one variant for another, or the decision to allow the substitution, can be made through the entire range of development times from acquisition time to run time. The substitution can be put into effect by a human actor or by a software monitor following a security policy, either randomly or in response to specific events in the environment.

Finally, an orthogonal consideration is the use of containment vessels to encapsulate components or subsystems within a virtual machine, to monitor and control interactions among components and subsystems in order to block attacks and protect vulnerable parts of a system. Figure 8 shows a screenshot in ArchStudio of a design-time architecture utilizing eight containment vessels, seven for individual components and connectors and the eighth for the group of components and connectors associated with the OS.

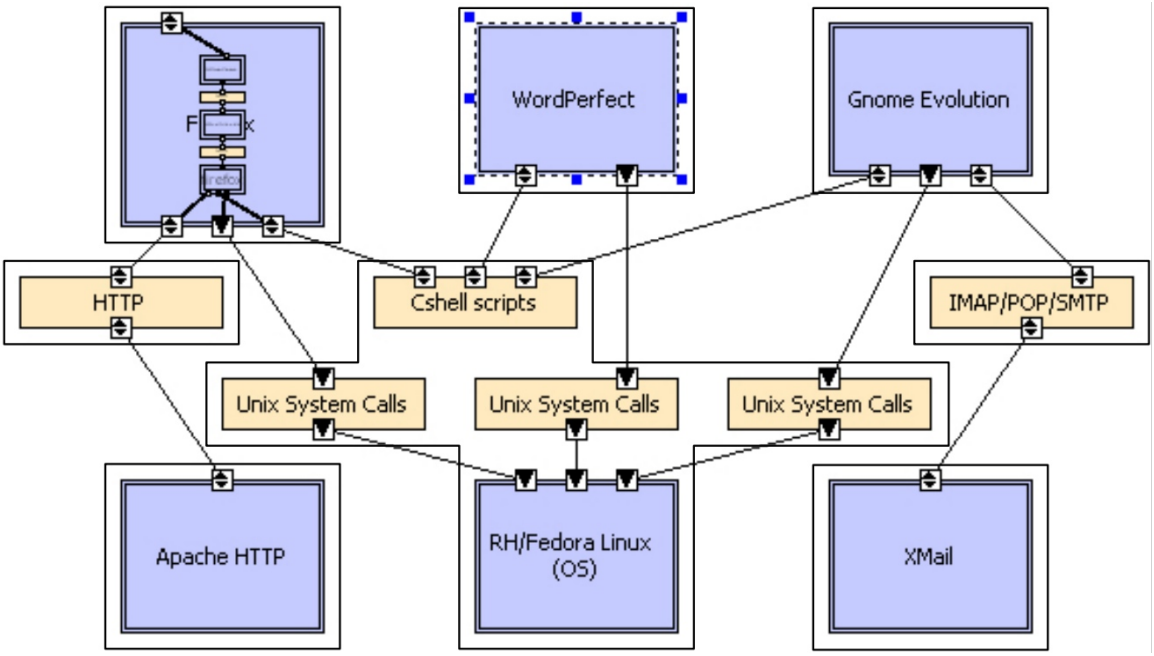

Figure 8. A Security Configuration Alternative for the Run-Time Configuration Instance That Encapsulates OA System Components and Connectors Within Different Virtual Machines (e.g., Using Xen (2012)).

For security, the GPL'd Fedora can employ the SELinux capabilities to restrict all shell/operating systems commands through mandatory access control and type enforcement (see Figure 8), while other components can all be contained within one (for minimal security confinement) or more (for increased security confinement on a per component basis) Xen-based virtual machines (again, See Figure 8). The interoperability of SELinux and Xen is now a common feature of many large Linux

system installations (e.g., Amazon.com now has more than 500,000 Linux systems running Xen; (SEL 2012; Xen 2012).

Discussion and Conclusions

Our goal in this study is to develop and demonstrate a new approach to address challenges in the acquisition of secure OA software systems. Program managers, acquisition officers, and contract managers will increasingly be called on to provide review and approval of security measures that are employed during the design, implementation, and deployment of OA systems. We seek to make this a simpler and more transparent endeavor. This requires security policies that are appropriate for review and approval during acquisition by people who may not be expert in the specifics of how best to insure that secure systems will result. Our view is to address this need by investigating how best to specify or model system security in ways that can accommodate security as a continuous process that must be supported throughout the system acquisition life cycle for OA systems (Scacchi & Alspaugh, 2008, 2011).

Our efforts reported here reveal that it is possible to employ a scheme through which complex OA systems can be designed, built, and deployed with alternative components and connectors into functionally similar system versions, in ways that allow for overall system security through the use of multiple security mechanisms. We described a scheme for how to realize and specify such OA system configurations in ways that are inherently compatible with existing security mechanisms, and this scheme does not assume that individual system elements must be secure before inclusion into the secured system's configuration. Central to our scheme is the incorporation of software product line concepts that are integrated with security mechanisms in a coherent way that is amenable to automated support and acquisition management. We also provided a case study that reveals where and how we specify a secure OA enterprise system product line in ways that can accommodate the diverse needs of software producers and developers, system integrators, users, and acquisition managers. What remains as an important next step for this line of research effort is to more fully articulate how to simply and transparently specify OA system security using streamlined security policies that utilize the kind of system security licenses we anticipate (Scacchi & Alspaugh, 2011), as well as designing and developing a prototype automated system that can support the modeling and analysis of OA system security policies, alternative version OA system configurations, and different OA security licenses.

References

[AAS09] Alspaugh, T. A., Asuncion, H., & Scacchi, W. (2009, August 31–September 4). Intellectual property rights requirements for heterogeneously licensed systems. In *Proceedings of the 17th IEEE International Requirements Engineering Conference (RE '09)* (pp. 24–33). Los Alamitos, CA: IEEE.

[AAS11] Alspaugh, T. A., Asuncion, H., & Scacchi, W. (2011, July). Presenting software license conflicts through argumentation. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*. Knowledge Systems Institute, Skokie, IL.

[ASA10] Alspaugh, T. A., Scacchi, W., & Asuncion, H. (2010, November). Software licenses in context: The challenge of heterogeneously licensed systems. *Journal of the Association for Information Systems*, *11*(11), 730–755.

[Bo06] Bosch, J. (2006, December). The challenges of broadening the scope of software product families. *Communications of the ACM, 49*(12), 41–44.

[Bo09] Bosch, J. (2009). From software product lines to software ecosystems. In *Proceedings of the 13th International Software Product Line Conference (SPLC '09)* (pp. 111–119). ACM, New York.

[BA05] Breaux, T. D., & Antón, A. I. (2005). Analyzing goal semantics for rights, permissions, and obligations. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE '05)* (pp. 177–188). Los Alamitos, CA: IEEE.

[BA08] Breaux, T. D., & Antón, A. I. (2008). Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering*, *34*(1), 5–20.

[CN01] Clements, P., & Northrop, L. (2001). *Software product lines: Practices and patterns.* New York, NY: Addison-Wesley.

[CIO10] Chief Information Officer, Department of Defense (DoD) Open Source Software (OSS) FAQ (2010). *Frequently asked questions regarding open source software (OSS) and the Department of Defense (DoD)*. Retrieved from http://dodcio.defense.gov/OpenSourceSoftwareFAQ.aspx

[DoD11] DoD (2011). *Department of Defense strategy for operating in cyberspace*. (2011, July). Retrieved from http://www.defense.gov/news/d20110714cyber.pdf

[FM11]Falliere, M., Murchu, L. O., & Chien, E. (2011, February). *W32.Stuxnet dossier, version 1.4*. Retrieved from http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[F04] Firesmith, D. (2004, January–February). Specifying reusable security requirements. *Journal of Object Technology*, *3*(1), 61–75.

[Fr10] Franz, M. (2010, September 21–23). E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the New Security Paradigms Workshop (NSPW '10)* (pp. 7–16). Retrieved from http://www.ics.uci.edu/~franz/Zurich/MassiveScaleDiversity.pdf

[Ga10] Garcia, P. (2010). Maritime C2 strategy: An innovative approach to system transformation. In *Proceedings of the 15th International Command & Control Research & Technology Symposium* (Paper 147). Retrieved from http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA526453

[Gi11] Gizzi, N. (2011). Command and control rapid prototyping continuum (C2RPC) transition: Bridging the valley of death. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1, pp. 127–154). Retrieved from http://www.acquisitionresearch.net

[GPL07] GPL (2007). *GNU General Public License, version 3*. Free Software Foundation. Retrieved from http://www.gnu.org/licenses/gpl.html

[H11] H, *Attack of the computer mouse*. (2011, June 29). *The H Online Security*. Retrieved from http://h-online.com/-1270018, accessded 1 April 2012.

[LSM98] Loscocco, P., Smalley, S., Muckelbauer, P., Taylor, R., Turner, S., & Farrell, J. (1998). The inevitability of failure: The flawed assumption of security in modern computing environment. In *Proceedings of the 21st National Information Systems Security Conference* (pp. 303–314), National Institute for Standards and Technology, Silver Springs, MD.

[NS87] Narayanaswamy, K., & Scacchi, W. (1987). Maintaining configurations of evolving software systems. *IEEE Transactions on Software Engineering*, *13*(4), 323–334.

[Navy10] Navy.mil. (2010). *PEO IWS releases open architecture contract guidebook update.* Retrieved from http://www.navy.mil/search/display.asp?story_id=53661, accessed 1 April 2012.

[SEL12] SEL, *SELinux on Xen*. (2012). Retrieved from http://wiki.prgmr.com/mediawiki/index.php/SELinux_on_Xen, accessed 1 April 2012.

[SJW11] Salamat, B., Jackson, T., Wagner, G., Wimmer, C., & Franz, M. (2011, July). Run-time defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing, 8*(4). 588-601.

[Saw11] Sawers, P. (2011, June 28). US Govt. plant USB sticks in security study, 60% of subjects take the bait. *TNW: The Next Web*. Retrieved from http://thenextweb.com/industry/2011/06/28/us-govt-plant-usb-sticks-in-security-study-60-of-subjects-take-the-bait.

[SA08] Scacchi, W., & Alspaugh, T. (2008). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (NPS-AM-08-036; Vol. 1, pp. 230–244). Retrieved from http://www.acquisitionresearch.net

[SA11] Scacchi, W., & Alspaugh, T. (2011). Advances in the acquisition of secure systems based on open architectures. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1, pp. 50–69). Retrieved from http://www.acquisitionresearch.net

[SA12] Scacchi, W., & Alspaugh, T. (2012). Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems, *Journal of Systems and Software*, 85(7), 1479-1494, July 2012.

[SBN11] Scacchi, W., Brown, C., & Nies, K. (2011, July). *Investigating the use of computer games and virtual worlds for decentralized command and control* (Final Report, Grant #N00244-10-1-006). University of California, Irvine, Institute for Software Research. Retrieved from http://www.ics.uci.edu/~wscacchi/ProjectReports/NPS-Reports/DECENT.pdf

[Se08] Seacord, R. (2008). *The CERT C secure coding standard*. New York, NY: Addison-Wesley.

[Sh11] Shrobe, H. (2011, November). *Secure computing systems*. Presentation at the Darpa Colloquium on Future Directions in CyberSecurity, Arlington, VA.

Retrieved from www.darpa.mil/WorkArea/DownloadAsset.aspx?id=2147484460

[Sm12] Smalley, S. (2012). *The case for Security Enhanced (SE) Android*. Retrieved from https://events.linuxfoundation.org/images/stories/pdf/lf_abs12_smalley.pdf

[SSL99] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., & Lepreau, J. (1999). The Flask Security Architecture: System support for diverse security policies. In *Proceedings of the Eighth USENIX Security Symposium* (pp. 123–139), USENIX Association, Berkeley, CA.

[STIG11] STIG, *Security technical information guide, Android 2.2 (Dell)*. Retrieved from http://iase.disa.mil/stigs/net_perimeter/wireless/smartphone.html, accessed 1 April 2012.

[Stux11] Stuxnet. (n.d.). In *Wikipedia*. Retrieved from http://en.wikipedia.org/wiki/Stuxnet , accessed 1 April 2012.

[SWZ12] Sun, K., Wang, J., Zhang, F., & Stavrou, A. (2012). SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity OSes. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium.* Internet Society, http://www.internetsociety.org/sites/default/files/P10_2.pdf, accessed 1 April 2012.

[Xen12] Xen, *Xen Hypervisor Project*. Retrieved from http://www.xen.org/products/xenhyp.html, accessed 1 July 2012.

[YC06] Yau, S. S., & Chen, Z. (2006). A framework for specifying and managing security requirements in collaborative systems. In *Proceedings of the Third International Conference on Autonomic and Trusted Computing (ATC 2006)* Springer, New York, (pp. 500–510).

Acknowledgments

Research described in this report was supported by grant #N447602-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School, and from grant #0808783 from the National Science Foundation. No review, approval, or endorsement implied.

THIS PAGE INTENTIONALLY LEFT BLANK

Exploring the Potential of Virtual Worlds for Decentralized Command and Control

Walt Scacchi, Craig Brown, & Kari Nies

Abstract

This report describes results from a research study investigating how virtual world (VW) concepts, techniques, and tools can be employed to create an online environment that supports experiments in Decentralized Command and Control (DCC). We refer to this project and the VW we have prototyped collectively as the DECENT project and system platform. DECENT is a platform for exercising and assessing the potential of a game-based VW approach to decentralized C2, as well as for comparing our efforts with others closely related. Overall, we find this effort gives rise to very promising results that point to additional opportunities and system extensions for new ways to consider the potential of decentralized approaches to C2 that merit further systematic investigation and experimentation. This report provides a description of the approach to prototyping and initially evaluating some of the potential of DCC systems based on VW technologies.

Overview

Decentralized Command and Control (DCC) is emerging as a new strategic thrust (DoD, 2012). DCC is envisioned as a new approach and model for how to organize and experience command and control systems, mission planning and scheduling processes, and physically decentralized user practices, using low-cost or free open source software technologies. DCC systems are anticipated to operate as virtual enterprises that are physically distributed but logically centralized. They are used at the edge of a multi-site organization, and thus can engage participants in different locations.

This report describes results from a research study investigating how virtual world (VW) concepts, techniques, and tools can be employed to create an online environment that supports experiments in DCC. We refer to this project and the VW we have prototyped collectively as the *DECENT* project and system platform. DECENT is a platform for exercising and assessing the potential of a VW-based approach to decentralized C2, as well as for comparing our efforts with others closely related. A companion paper further describes how DECENT has been used to support the creation and experimentation with C2 mission planning games (Scacchi, Brown, & Nies, 2012). Overall, we find this effort gives rise to very promising results that point to additional opportunities and system extensions for new ways to consider the potential of decentralized approaches to C2 that merit further systematic investigation and experimentation.

Our choice to employ VW technologies is in part influenced by the growing pervasiveness of such technologies, their availability as open source software in user

modifiable forms, and their widespread use by a new generation of online computer users who may see/anticipate that such technologies will become ubiquitous in future enterprise settings.

Next, our interest is not to simply replicate or mirror existing C2 systems, nor their traditional patterns of usage. Such usage generally assumes both centralized, hierarchical organizational authority and centralized location of users. Instead, our interest is to explore the alternative space where decentralized approaches to organizational decision-making and workplace location (e.g., top-down but physically dispersed versus peer-to-peer and physically dispersed) may be subject to experimental variance and study.

We similarly identify and compare a small set of related technologies that could be compared to the efficacy of the VW technologies that we employ (*OpenSim* [2012], an open source software toolkit for building, navigating, and socially interacting in VWs). OpenSim provides many interesting affordances, some of which are common to most VWs. But it is these affordances that merit further study. Understanding the potential for how VWs may be designed, built, deployed, and evolved seems to be a significant opportunity area for further study. In addition, there is still a need to determine how best to evaluate and compare the efficacy of VWs that seek to mirror physical sites or physically located human problem solving and social interaction. There is also a need to evaluate and compare the efficacy of alternative VW and computer game development technologies, whether open source software, or proprietary, closed source software. Last, we also find that decentralized VW-based approaches may offer the potential to substantially reduce the cost and dramatically shorten the time to design, build, and deploy C2 systems that embrace new generations of low-cost, mobile technologies that future C2 workforces may expect, whether for use in physical or virtual/cyberspace worlds. So much remains to be studied, and the time for appropriate and realistic research investments is at hand. In the near term, such research is likely to still be considered risky, but the longer-term benefits may most quickly arise and be demonstrated through such near- to mid-term research investments. This future opportunity is now at hand.

Developing a DECENT Prototype

In the effort described here, we have prototyped a computer game and virtual world (CGVW) environment we call the DECENT project. Our efforts here represent a substantial departure from current C2 practice, and thus do not seek to primarily provide an incremental improvement to centralized C2 efforts. However, our research is informed by such efforts, like the C2 Rapid Deployment Continuum (C2RPC) highlighted in Figure 1, as they are critical to enhancing and demonstrating upgrades to current C2 operations which have high consequence (Garcia, 2010; Gizzi, 2011).

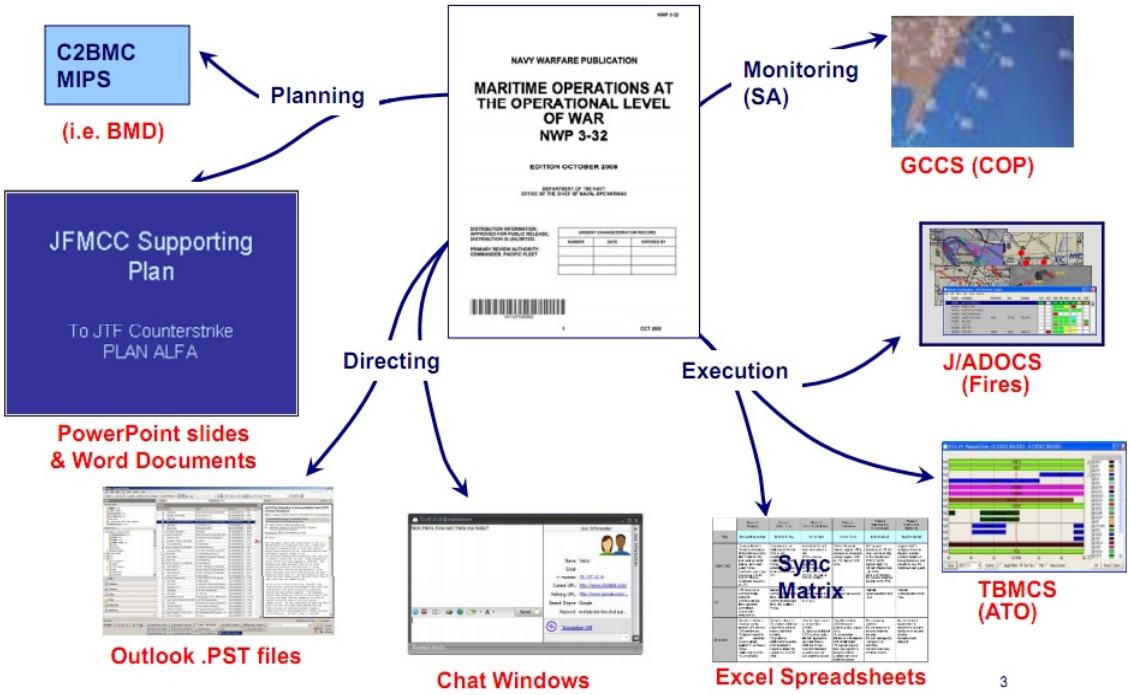

Figure 1. Common Information Objects and Software Applications That May Be Involved in C2 Operations/Tasking, as Identified in Publicly Accessible C2RPC (2010) Materials

In such settings, C2 operations/tasks entail the creation, update, and sharing/presentation of information reports for C2 decision-making purposes, which may include mission plans and resource assignment schedules expressed as timelines or spreadsheets, for example.

However, as our efforts represent a basic research investigation, we can pursue more risky pathways and edgy alternatives that may or may not yield significant advances. Furthermore, our attention is directed to technologies that enable network-centric, decentralized "edge" approaches to C2 (Albert & Hayes, 2003). Consequently, our goal is to advance scientific and technical knowledge for how decentralized C2 might be put into practice in the future, especially with regard to future workforces who may have grown up playing computer games and/or exploring virtual worlds.

Developing Virtual Worlds of Physical Places

In order to achieve the highest quality for DECENT, development began by examining existing C2 structures and identifying key features to replicate. Ground-based C2 facilities usually have several large screens on several walls, used for the display of two types of information: shared information that must be available to several personnel simultaneously, and key information that is either higher priority or more topically relevant. These C2 elements can be seen in Figure 2. While these large public displays are viewable by anyone in the command room, most C2 workers also have their own personal computer (often with multiple displays), housed on desks with assorted papers, files, schedules, and so forth.

Figure 2. Photograph of a Physical C2 Facility at Hanscom Air Force Base [Hanscom 2010].

Note in Figure *2.* both shared (wall screens) and private resources (tabletop displays) can be seen, with other mission tasking resources.

Systems used in C2 facilities require a range of software, and information from one user may affect the relevance of another's information. Personnel are thus typically organized in a way that optimizes communication, with the most frequent communication being between neighbors. Nevertheless, spaces in C2 facilities often seem crowded and cluttered, despite the need for efficiency.

Virtual World Space

Using images like the one in Figure 2, we have created a virtual C2 world. Taking advantage of the malleable nature of virtual worlds, we can make C2 rooms with more space and a less cluttered or cramped appearance. All of the important aspects and features common to C2 facilities have been faithfully and dutifully recreated. Simple tables contain monitors, input devices, paperwork (which can be modified or made interactive), and speakerphone boxes. The virtual workspace can be filled with clipboards, pens, soda cans, coffee mugs, and other cosmetic/non-functional objects. This variability allows each DECENT implementation to be customized by whoever is running the training system. Chairs allow the user's avatar to be anchored to (i.e., seated) within given workspace, and two monitors act as that user's private information displays, such as DECENT training game data specific to that user. These details can all be seen in Figure 3.

Two of DECENT's walls have large displays for the display of public information, including the main screen for the DECENT training game. These images may be easily changed, and can be used to display streaming video, as well as static images and the DECENT mission planning training game. Due to the nature of virtual worlds, the modeling of bulky physical items necessary for actual C2 centers, such as computation and data reduction servers, PC boxes, and cabinets can be deferred to a later time or ignored entirely.

Figure 3. Perspective View of User-Controlled Avatars in a C2 Mirrored VW Operating in DECENT

Note. The wall display in the upper right corner is an embedded video stream from a remote server.

Comparing Physical Places and Virtual Worlds

The strategy we have investigated in the DECENT environment is to prototype a mirrored virtual world for C2 that resembles and may operate like the physical world C2. In this way, we seek to explore and examine when/how the similarities and differences between the two can reveal potentially significant insights, opportunities, or advantages that one may pose over the other. For example, in studies with VWs at the Naval Postgraduate School's Center for the Edge, there has been sustained study examining how hypotheses about different models of team organization or theories of management might affect the course and outcome of play in the ELICIT multiplayer online counterterrorism intelligence game (Bergin et al., 2010; Hudson & Nissen, 2010, 2011; Wynn, Ruddy, & Nissen, 2010). Among other things, these studies seek to investigate the efficacy of organizational form, team play, and outcome in the ELICIT game when played in a physical setting (a large unadorned meeting room) in comparison to a virtual setting, seen in Figure 4.

Figure 4. On-Screen View of a Virtual Meeting Room That Mirrors Common Meeting Rooms, Used to Study Team-Oriented ELICIT Game Play
(Hudson & Nissen, 2010)

Studies by Bergin et al. (2010) suggest that decision-making performance in physical and virtual worlds can favor the physical. This may be due to the environmental richness and tacit knowledge affordances that familiar work spaces and co-worker gestures/gazes provide, compared to the paucity of similar affordances in a VW. However, DECENT VW may offer other benefits like low cost, appropriateness for large-scale training, and absence of a centralized (potentially vulnerable) C2 physical center. Elsewhere, other research groups have been experimenting with the creation of mirror worlds that intermix physical world sites, with VW interfaces and navigational and interactive controls to devices in the physical site. One noteworthy example is the effort by Back, et al. (2010), who modeled a physical factory (the TCHO chocolate factory located in San Francisco), as part of their efforts at Fuji Xerox Palo Alto Laboratory (FXPAL). Their VW system includes both desktop PC–based and smartphone-based software clients that allow a user to navigate the VW space, and to enable/disable designated sensors (Web cameras) located in the physical site, and thus demonstrate the potential to remotely control or monitor devices in the physical site though the VW client interface, shown in Figure 5.

The FXPAL system thus demonstrates the potential for mixed reality applications that span and interlink physical world sites that are mirrored in a VW. Their efforts, in turn, can be compared with one of our earlier efforts that focused on the modeling and simulation of semiconductor fabrication processes and the diagnosis of manufacturing devices for training technicians (Scacchi, 2010).

Figure 5. Smart-Phone (iPhone)–Based Views for Monitoring and Controlling Devices in a Physical Factory
(Back, Kimber, et al., 2010)

However, this effort was based on abstractions of semiconductor and nanotechnology fabrication facilities on site at University of California, Irvine, but generalized into configurations that were suggested by the project sponsor at Intel Corporation (Scacchi, 2010).

Platform for VW Development: OpenSim

Due to its ease of use and rapid development capabilities, DECENT is currently implemented in OpenSim, an open-source workalike of the closed-source *Second Life* VW platform. Second Life (2012) is the current market leader in rapid virtual world development and operation, with a high level of design flexibility and built-in tools for easy environment creation and maintenance. This makes it and OpenSim ideal for the creation of prototypes. Using OpenSim has allowed us to rapidly create a functional C2 VW analog, and populate it with users for concept prototyping, testing, and experimentation. The degree of design freedom provided by OpenSim has allowed DECENT to evolve from a promising concept, into a functional training, experimentation, or demonstration environment.

While in OpenSim, DECENT has the potential to seamlessly interact and crossover with other currently existing military projects, such as the Military Open Simulator Enterprise Strategy (MOSES; 2012) combat training environment and the Naval Underwater Warfare Center (NUWC) campus (Aguiar & Monte, 2011). Adding DECENT to either MOSES or NUWC is a matter of adding a new region and importing the DECENT assets, or establishing a hypergrid connection (described later in this report, and in Lopes, 2011) or federation between these disparate virtual worlds. This could be done as many or as few times as needed, and each instance of DECENT would act

independently. The person responsible for MOSES and NUWC, Douglas Maxwell, has stated, “All of [OpenSim's] features are desirable for the new virtual trainers needed to meet the changing situation demands on modern warfighters” (Neville, 2011). Last, using the OpenSim/Second Life modeling tools, it is possible to create relatively complex objects, as well as associating behavioral scripts (using LSL) to enable rich animated behaviors to be associated with different objects.

SecondLife Versus OpenSim

While OpenSim is an open source project based on the Second Life platform, each has its own strengths and weaknesses, as described by Korolov (2011):

Cost: The cost of running a Second Life server is $295 each month, as it must be hosted on a Linden Labs server. OpenSim can be acquired, installed, and run for free. It can be installed on a dedicated server to host one or more VWs which can also be configured into a local/wide-area grid through the OpenSim Hypergrid (Lopes, 2011), described in this section. Reasonably simple versions of OpenSim are available in download formats that allow for distribution via portable USB flash storage (thumb) drives, which means their potential to become widespread and disruptive is emerging. We currently host five dedicated OpenSim servers at UCI.

Users: Second Life is a community, so it comes with a large established base of users from all conceivable backgrounds. OpenSim has access to smaller groups of existing users if connected to existing OpenSim servers, but is more often run as a stand-alone server, only used by those whom the administrators give access to.

Stability: Second Life maintains high stability for functions used by the largest majority of its users, but is unreliable for mission-critical operations due to problems with voice-chat and reoccurring connectivity issues. OpenSim tends to have much more stable connectivity and voice-chat, (and now integrated instant messaging support within and across worlds) due to the smaller number of users and lower required bandwidth.

Asset Ownership: Linden Labs retains the rights to all assets created in Second Life (but not content uploaded to it), regardless of who created it; users pay for a license to use Second Life and modify the contents of a region, but gain no ownership of actual content. Furthermore, Linden Labs reserves the right to revoke access to Second Life. In contrast, owners of OpenSim servers determine the use policies and ownership of their servers, as well as control access to their servers.

Scripting: Second Life uses the Linden Scripting Language (LSL), a domain-specific scripting language. OpenSim supports LSL, but can be modified to support many other scripting languages, including JavaScript and Lua (which is a popular scripting language employed by many computer games, including *World of Warcraft*).

The OpenSim Hypergrid

In addition to the reasons laid out about by Korolov (2011), we have decided to work with OpenSim because of its connections to the OpenSim *Hypergrid* (2012). The

Hypergrid is a system used to connect one OpenSim server to others, and allows for the seamless transfer of avatars between any of these interconnected OpenSim worlds, as depicted in Figure 6. Some large worlds span multiple servers. The virtual worlds are connected to each other via virtual world hyperlinks, similar to the links between Web pages. The virtual world hyperlinks are places in the virtual worlds that act as doorways or entry/exit points for other worlds. To access one, a user simply moves his avatar to the VW hyperlink and activates it. VW hyperlinks could be used to connect DECENT to existing OpenSim-based military training systems, such as the hypergrid for the MOSES Server Map (2012) facility, as displayed in Figure 7.

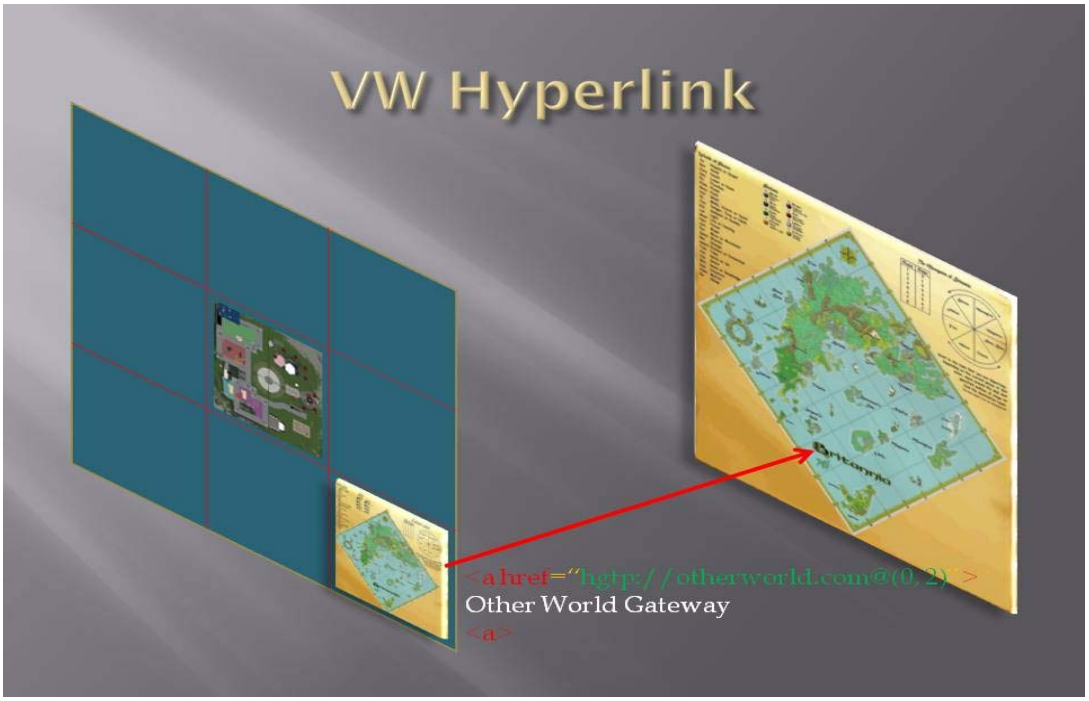

Figure 6. OpenSim Hypergird Supports User/Avatar Teleportation to Move From One VW Region to Another VW Region, Possibly on a Different Networked Server (Lopes, 2011).

Under-Explored Topics for DECENT

To no surprise, there are many other topics for investigation that were beyond the scope of resources available for us to explore. For example, most CG software technologies offer little or no ready support for integration of external application programs or other software components. So our efforts to model and simulate a decentralized C2 virtual world make no attempt to undertake such integration studies.

Figure 7. Overview of the *MOSES Hypergrid* and Server Assignment Map Used in the Military OpenSim Enterprise Strategy by the U.S. Army
(MOSES Server Map 2012)

Note. Some hypergrid cells are empty, indicating available capacity for future development of new virtual worlds. Map shown in Figure as of September 2011.

Next, the underlying software architectures of CGVW are rarely disclosed or made open, even when realized using open source software (OSS) components. So it is a major technical challenge to evaluate, assess, or compare at a deep technical level what architectural choices or trade-offs have been made in designing, building, and/or deploying an operational game-based VW system. In simple terms, this makes comparing OSS VW technologies like Delta3D (see http://www.delta3d.org/), OpenSim, and any of dozens of OSS game engines accessible on the Web (e.g., via search at SourceForge.net) impractical at present. Thus, there is a basic research need to develop open architecture (OA) frameworks for specifying CGVW systems (Scacchi & Alspaugh, 2008).

Similarly, the topic of *securing a game-based VW for military C2 applications* is a major concern. VWs like DECENT are envisioned, developed, and extended as an open architecture system (Scacchi & Alspaugh, 2008). Recent advances in developing architectural level security schemes for designing, building, and deploying open architecture software systems are relevant and readily applicable to VWs applications, as well as Web-based system architectures, such as those for the C2RPC (2010). For example, Scacchi and Alspaugh (2011) have developed and demonstrated a conceptual approach based on existing research technologies that can be used to specify, model,

and analyze the security of an OA system with secure/contained elements, as suggested in Figures 8 and 9.

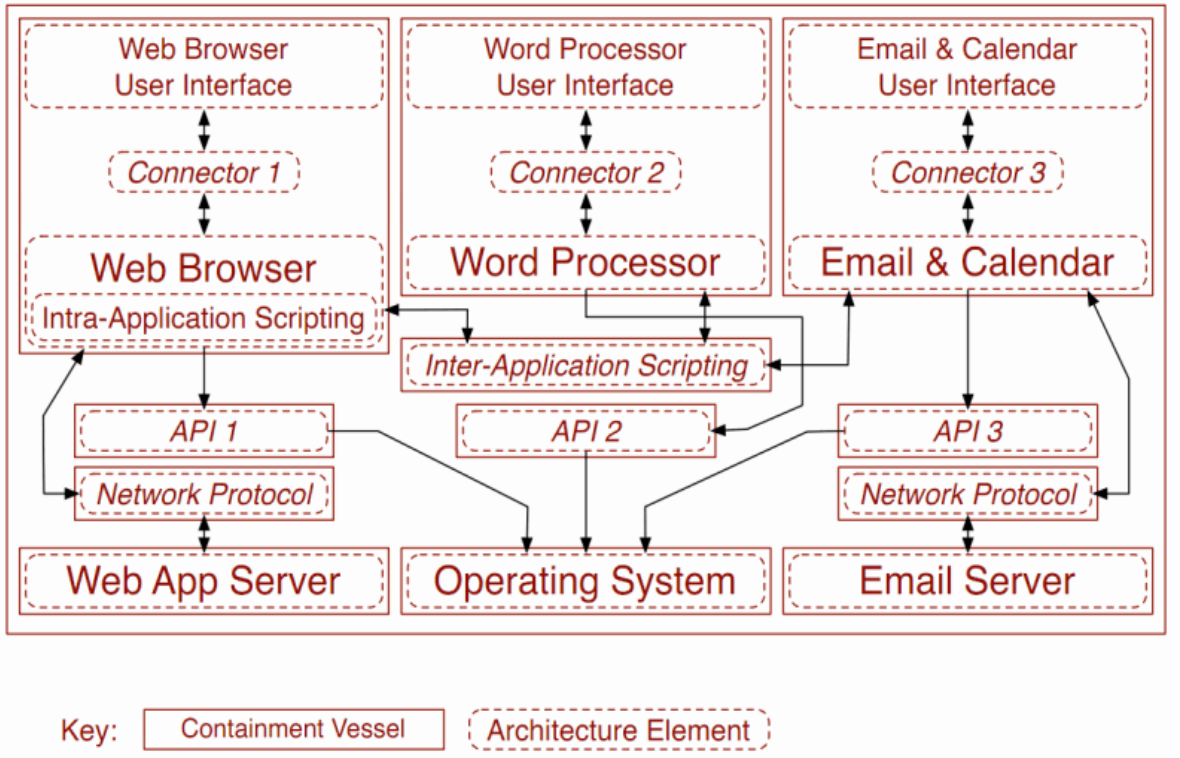

Figure 8. Secure Architectural Design for Generic Software Applications Common to C2 Systems
(Scacchi & Alspaugh, 2011)

Last, most CG software technologies provide very basic security mechanisms, and thus are quite amenable to remote attacks, penetration, and possible code injections. Furthermore, game-based VW may allow for new modes of malware that may enable activities including avatar impersonation or remote control (e.g., who/what is controlling this avatar, and with what authorization?) and other ill-defined vulnerabilities. So CGVW technologies should not be considered for deployment purposes until more robust security capabilities are in place, tested, validated, monitored, and evolved (cf. Scacchi & Alspaugh, 2011), such as those in the C2RPC (Garcia, 2010; Gizzi, 2011). However, they may be appropriate for experimentation with future C2 system architectures that may include Web-based and game-based VW software elements, that may be accessible from smartphones, as well as open to access, monitor, and control physical devices and sensors deployed in physical world settings, whether on land, sea, air, or space.

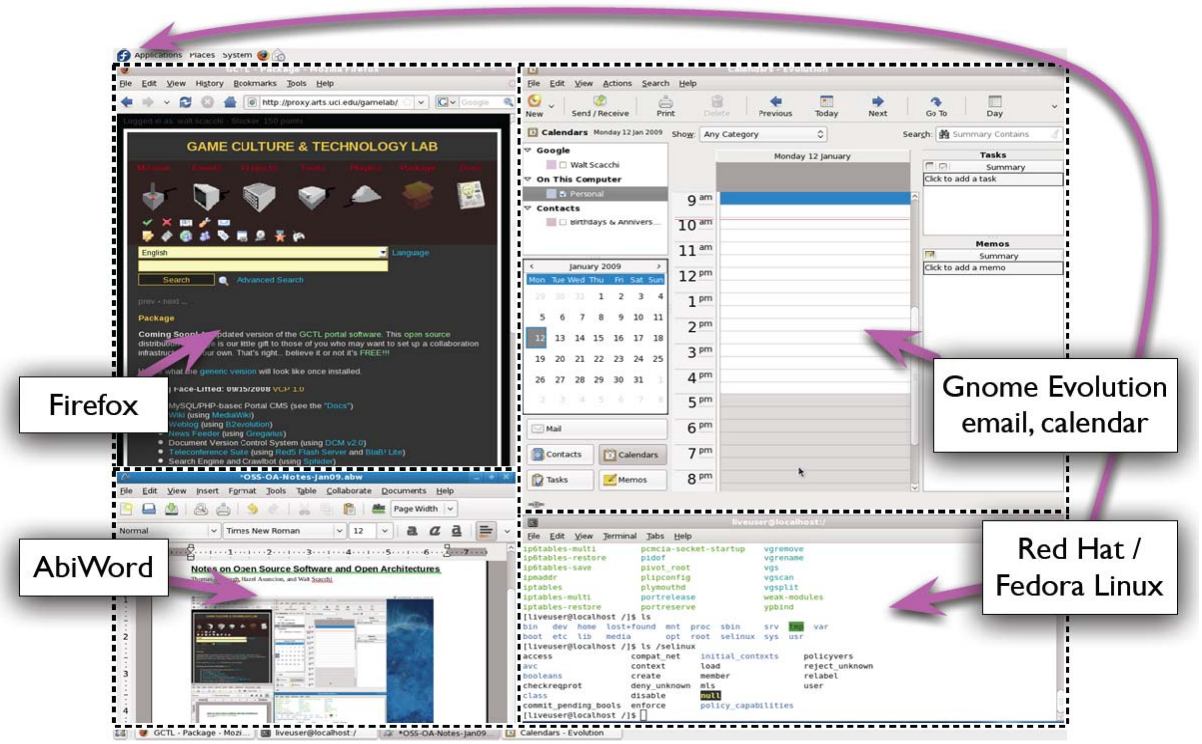

Figure 9. Sample Run-Time Software Application Deployment for Generic Software Applications Available for Use in C2 Systems
(Scacchi & Alspaugh, 2011)

Conclusions and Recommendations for Future Study

This report seeks to describe and document the results of a small-scale, one-year research study that investigates how virtual world concepts, techniques, and tools can be employed to support experimental/prototyping efforts for command and control applications. We reported on our efforts to investigate and prototype a VW we called DECENT as a platform for exercising and assessing the potential of a VW-based approach to decentralized C2, as well as to compare our efforts with others closely related. A companion paper provides additional details and supporting materials on DECENT (Scacchi et al., 2012), and to our results presented here. Overall, we found this effort gave rise to very promising results that point to additional opportunities and system extensions for new ways to consider the potential of decentralized approaches to C2 that merit further systematic investigation and experimentation.

We similarly identified and compared a small set of related technologies that could be compared to the efficacy of the VW technologies that we employed (OpenSim, an open source software toolkit for building, navigating, and socially interacting). OpenSim provides many interesting affordances, some of which are common to most VWs. In particular, the potential exists for creating and disseminating free open source software–based versions of DECENT or similar DCC software systems using pocket-friendly mobile storage devices (e.g., USB flash storage) that can then be installed on most PCs. The DECENT prototype thus demonstrates a transformative reduction in the

cost of rapidly creating and deploying C2 systems that can support DCC, as well as supporting the potential to integrate and control cyber space applications and remote commands.

Understanding the potential for how VWs may be designed, built, deployed, and evolved seems to be a significant opportunity area for further study. In addition, there is still a need to determine how best to evaluate and compare the efficacy of VWs that seek to mirror physical sites or physically located human problem solving and social interaction. There is also a need to evaluate and compare the efficacy of alternative VW and computer game development technologies, whether open source software, or proprietary, closed source software.

Last, much remains to be studied, and the time for appropriate and realistic research investments is at hand. In the near term, such research is likely to still be considered risky, but the longer-term benefits may most quickly arise and be demonstrated through such near- to mid-term research investments. This is the future opportunity now at hand.

References

Aguiar, S., & Monte, P. (2011). Virtual worlds for C2 design, analysis, and experimentation. In *Proceedings of the 16th International Command & Control Research & Technology Symposium.*. Retrieved from http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA547157

Albert, D. S., & Hayes, R. E. (2003). *Power to the edge: Command and control in the information age*. Washington, DC: Command and Control Research Program. Retrieved from http://www.dodccrp.org/files/Alberts_Power.pdf

Back, M., Kimber, D., Rieffel, E., Dunnigan, A., Liew, B., Gattepally, S., Foote, J., Shingu, J., & Vaughan, J. (2010). The virtual chocolate factory: Mixed reality industrial collaboration and control. In *Proceedings of the International Conference on Multimedia (MM '10)* (1505–1506). New York, NY: ACM.

Bergin, R., Adams, A., Junior, R., Hudgens, B., Chinn Yee Lee, J., & Nissen, M. (2010). Command & control in virtual environments: Laboratory experimentation to compare virtual with physical. In *Proceedings of the 15th International Command & Control Research & Technology Symposium* (Paper 075). Retrieved from http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA524136

C2RPC. Command and Control Rapid Deployment Continuum (2010). *Command and control rapid deployment continuum overview*. Retrieved from http://www.afcea-sd.org/wp-content/uploads/2010/12/YoungAFCEA_C2RPC.pdf

Department of Defense (DoD). (2012, January 17). *Joint operational access concept (JOAC), version 1.0*. Retrieved from http://www.defense.gov/pubs/pdfs/JOAC_Jan%202012_Signed.pdf

Garcia, P. (2010). Maritime C2 strategy: An innovative approach to system transformation. In *Proceedings of the 15th International Command & Control*

Research & Technology Symposium (Paper 147). Retrieved from http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA526453

Gizzi, N. (2011). Command and control rapid prototyping continuum (C2RPC) transition: Bridging the valley of death. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1, pp. 127–154). Retrieved from http://www.acquisitionresearch.net

Granlund, R., Smith, K., & Granlund, H. (2011). C3 conflict: A simulation environment for studying teamwork in command and control. In *Proceedings of the 16th International Command & Control Research & Technology Symposium.*. Retrieved from http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA547134

Hanscom Air Force Base (2010). Hanscom Air Force Base Air and Space Operations Center, http://www.hanscom.af.mil/shared/media/photodb/web/web_030314-F-9999G-003.jpg , accessed 1 April 2010.

Hudson, K., & Nissen, M. (2010). Command & control in virtual environments: Designing a virtual environment for experimentation. In *Proceedings of the 15th International Command & Control Research & Technology Symposium* (Paper 052). Retrieved from http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA524141

Hudson, K., & Nissen, M. (2011). Understanding the potential of virtual worlds in improving C2 performance. In *Proceedings of the 16th International Command & Control Research & Technology Symposium.* Retrieved from http://www.dtic.mil/dtic/tr/fulltext/u2/a546920.pdf

Korolov, M. (2011). *Second Life vs. OpenSim*, Retrieved from http://www.hypergridbusiness.com/2011/05/second-life-vs-opensim/

Lopes, C. V. (2011, September–October). Hypergrid: Architecture and protocol for virtual world interoperability. *IEEE Internet Computing*, *15*(5), 22–29.

MOSES. (2012). *Military open simulator enterprise strategy*. Retrieved from http://fvwc.army.mil/moses/. Currently located at http://brokentablet.arl.army.mil/index.html

MOSES Server Map. (2012). Image Retrieved from http://fvwc.army.mil/moses/server-allocation-map/. Current MOSES server map at http://107.7.21.240/map/

Neville, J. (2011, May 21). Army extends MOSES to other researchers. *Hypergrid Business*. Retrieved from http://www.hypergridbusiness.com/2011/05/army-extends-moses-to-other-researchers/

OpenSim. (2012). *The Open Simulator Project*. Retrieved from http://opensimulator.org/

OpenSim Hypergrid. (2012). *The OpenSimulator Hypergrid.* Retrieved from http://opensimulator.org/wiki/Hypergrid

Scacchi, W. (2010). Game-based virtual worlds as decentralized virtual activity systems. In W. S. Bainbridge (Ed.), *Online worlds: Convergence of the real and the virtual* (Human–computer interaction series; pp. 225–236). London, UK: Springer-Verlag London Limited.

Scacchi, W., & Alspaugh, T. (2008). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (NPS-AM-08-036; Vol. 1, pp. 230–244). Retrieved from http://www.acquisitionresearch.net

Scacchi, W., & Alspaugh, T. (2011). Advances in the acquisition of secure systems based on open architectures. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1, pp. 50–69). Retrieved from http://www.acquisitionresearch.net

Scacchi, W., Brown, C., & Nies, K. (2012, June). Exploring the potential of computer games for decentralized command and control. In *Proceedings of the 17th International Command & Control Research & Technology Symposium.* Retrieved from http://dodccrp.org/events/17th_iccrts_2012/post_conference/papers/104.pdf

Scacchi, W. et al. (2011, February). *The future of research in computer games and virtual worlds*. NSF Workshop Report. (Technical report UCI-ISR-12-8). Retrieved from http://www.isr.uci.edu/tech_reports/UCI-ISR-12-8.pdf

Second Life. (2012). *What is Second Life?* Retrieved from http://secondlife.com/whatis/

Wynn, D., Ruddy, M., & Nissen, M. (2010). Command & control in virtual environments: Tailoring software agents to emulate specific people. In *Proceedings of the 15th International Command & Control Research & Technology Symposium* (Paper 019). Retrieved from http://www.dtic.mil/dtic/tr/fulltext/u2/a524166.pdf

Acknowledgments

The research described in this report was supported by grant #N00244-10-1-0064 from the Center for the Edge Power and grant #00244-12-1-0004 from the Acquisition Research Program, both at the Naval Postgraduate School, Monterey, CA. No endorsement, review, or approval implied.

THIS PAGE INTENTIONALLY LEFT BLANK

Software Licenses, Open Source Components, and Open Architectures

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

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 application program interfaces (APIs). The emerging challenge is to realize the benefits of openness when components are subject to different copyright or property licenses. In this chapter, we identify key properties of OSS licenses, present a license analysis scheme to identify license conflicts arising from composed software elements, and apply this scheme 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 as a technique for aligning such architectures with enterprise strategies supporting open systems.

Introduction

A substantial number of enterprises and independent software vendors are adopting a strategy in which software-intensive systems are developed with open source software (OSS) components or components with open application program interfaces (APIs). It has been common for both independent and corporate-sponsored 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 homogeneously granted and the system has a well-defined OSS license, has been the dominant practice and continues to this day.

However, we more and more commonly see a different enterprise software configuration, in which the components of an enterprise 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 sub-licensed, if the corresponding obligations are met (Alspaugh, Asuncion, & Scacchi, 2009). 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 chapter 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 chapter seeks to explore and answer the following kinds of research questions:

- What license applies to an OA enterprise system composed of software components that are subject to different licenses?
- How do alternative OSS licenses facilitate or inhibit the development of OA systems for an enterprise?
- How should software license constraints be specified so it is possible for an enterprise to automatically determine the overall set of rights and obligations associated with a configured enterprise software system architecture?

This chapter 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 enterprise system configuration.
In the remainder of this chapter, 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 some issues raised by our work.

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. Open Source Initiative [OSI], 2011), 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 (and why) OSS and open APIs are located within the system architecture; (b) how OSS and open APIs are implemented, embedded, or interconnected in the architecture; (c) whether the copyright (Intellectual Property) licenses assigned to different OSS components encumber all or part of a software system's architecture into which they are integrated; and (d) whether alternative architectural configurations and APIs exist that may or may not produce an OA (cf. Antón & Alspaugh, 2007; Scacchi & Alspaugh, 2008). Subsequently, we believe that such ambiguity 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 (Scacchi, 2002; Scacchi 2009), or what architecture style (Bass, Clements, & 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 (Alspaugh, Scacchi, & Asuncion, 2010; Scacchi & Alspaugh, 2008).

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 the possible mix of software elements that 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 et al., 2003).

- *Software source code components*—These include: (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 & Churchill, 2006]).
- *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).
- *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 & Oberndorf, 2001).
- *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, & Dahmann, 2000), as are CORBA, Microsoft's .NET, Enterprise Java Beans, and LGPL libraries.
- *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 & Taylor, 2002) for Web-based client–server applications, or may represent an original or ad hoc architectural pattern (Bass et al., 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 (ADL) and ADL-based support tools (Bass et al., 2003; Medvidovic, Rosenblum, & Taylor, 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 listed in this section, as well as 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 Figure 1) 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 Figure 1) 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.

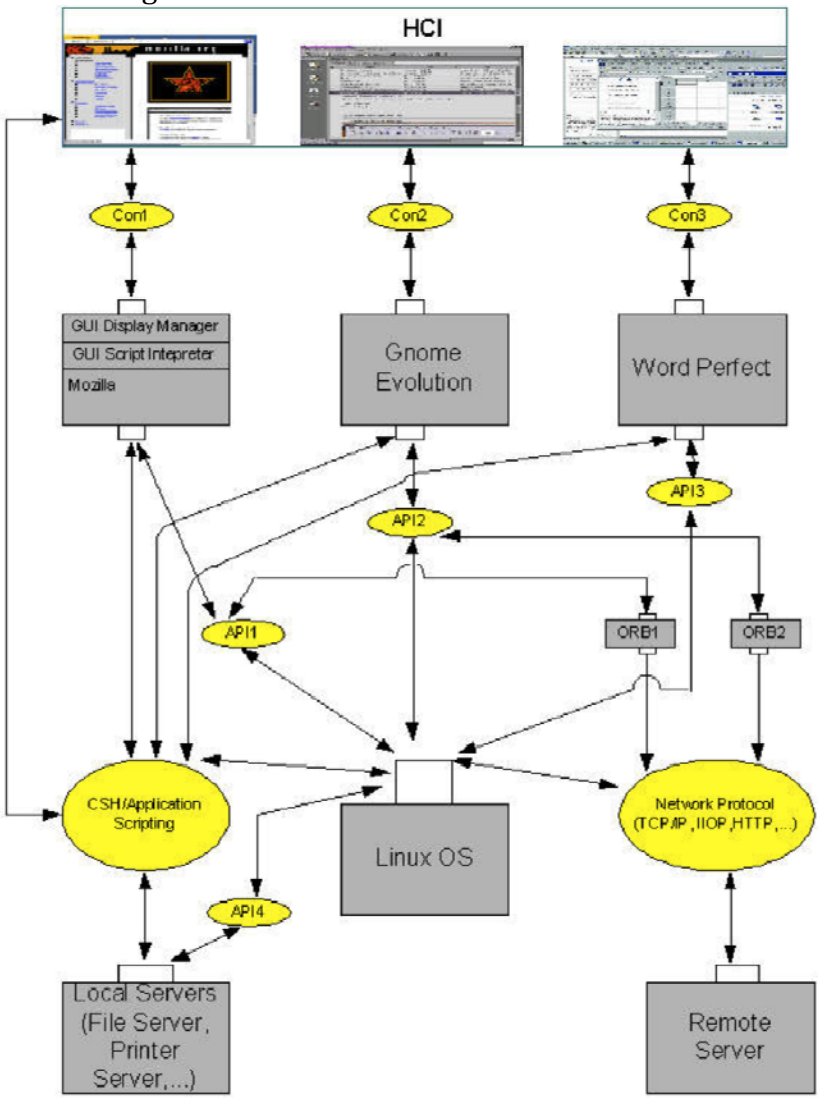

Figure 1. An Enterprise Software System Architecture

Note. This figure depicts components (grey boxes), connectors (ellipses), interfaces (small boxes on components), and data/control links.

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 et al., 2008; OSI, 2011; Rosen, 2005; St. Laurent, 2004).

Software licenses may be grouped into four general categories, listed in Table 1. OSS licenses are classified as permissive, reciprocal, and propagating; all propagating licenses of which we are aware are also reciprocal, but most reciprocal licenses are not propagating. End-user license agreements (EULAs) and terms of service (TOSs) for commercial software are typically proprietary and do not grant the OSS rights of copying, source code availability, modification, and distribution.

Table 2. Types of Software Licenses

License Type	Also known as	Examples	Characterized by
Permissive	Academic	Apache, BSD, MIT	Many rights; few obligations
Reciprocal	Copyleft	MPL, LGPL	Many rights; obligations on derivative works
Propagating	Strong Copyleft	GPL, AGPL	Many rights; obligations on "nearby" works
Proprietary		CTL, EULAs, TOSs	Few rights

Typical rights and obligations include the following:

A right to perform an action: "... each Contributor hereby grants to You a ... copyright license to reproduce ... the Work ... in Source ... form" (source: Apache License 2.0).

A right to not perform an action: "In no event shall the authors or copyright holders be liable for any claim, damages or other liability" (source: MIT License).

An obligation to perform an action: "You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License" (source: GPLv2 License).

An obligation to not perform an action: "Neither the name of the <organization> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission" (source: BSD 3-Clause License).

OSS licenses typically grant the right to copy, modify, and distribute source and binary code, while proprietary licenses typically grant only the right to possess one, or a limited number of, binary copies, or analogous rights to connect to a service, and often explicitly disallow modification or distribution. OSS licenses typically impose an obligation to retain copyright and license notices unmodified. Reciprocal licenses typically impose an obligation to publish source code of modified versions under the same license (the reciprocal obligation); propagating licenses are characterized by obligations to publish "nearby" software, under varying definitions of "nearby" that

range over software statically linked, dynamically linked, or other means of combining. Virtually all software licenses include disclaimers of liability and warranty, and newer licenses often include various provisions for patent rights.

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, 2011). Software components are now available subject to licenses such as the General Public License (GPL), Affero General Public License (AGPL), Mozilla Public License (MPL), Apache Public License, (APL), permissive licenses (e.g., BSD, MIT), Creative Commons, and Artistic. 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 chapter.

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, publish, distribute, sub-license, 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 (permissive licenses); the reciprocal obligation to publish source code you modify under the same license (MPL); or the propagating obligation to publish under GPL all source code for a work "based on the Program" where the "Program" is GPL'd software (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 permissive licenses, if you retain the copyright notice, list of license conditions, and disclaimer, then you can use, modify, merge, sub-license, and so forth. 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 advice begins to seem inevitable (cf. Fontana et al., 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 design-time 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.

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 time, build time, and run time, consistent with the different concerns and requirements that apply at each phase (cf. Scacchi & 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, objects*> for expressing a right or obligation. The *actor* is the "licensee" or "licensor" 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; 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 *objects* completing the description. We specify objects separately from the action in order to minimize the set of actions and to simplify the formalization of relations among rights and obligations. Objects are specified by parameters ranging over specified types, represented here by names in {}. An obligation's objects may also be specified by parameters, in [], bound to objects of the appropriate type for the corresponding right. Finally, some licenses are parameterized, represented by names in {{}}. See our previous work for more specifics (Alspaugh et al., 2009; Alspaugh et al., 2010; Alspaugh, Asuncion, & Scacchi, 2011). 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 permissive BSD software license. Note that the first right is granted without corresponding obligations.

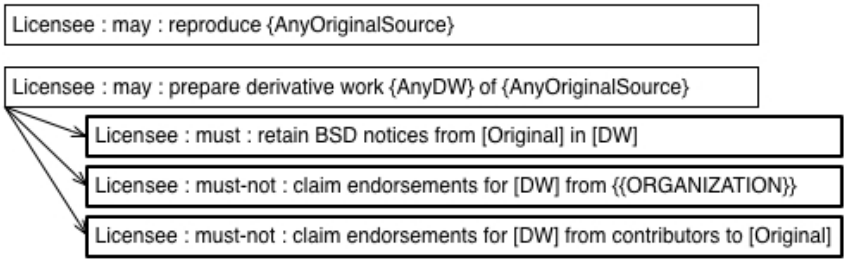

Figure 2. A Portion of the BSD License Tuples

The process of expressing licenses with tuples is manual, with the majority of the effort spent identifying each action and placing it in an ontology of actions from all the licenses of interest. The ontology is needed for reasoning about the actions; from it the subsumption relationship between any two actions can be automatically determined. Some actions, such as those for the exclusive copyright rights, are widely shared among licenses; others, often those for obligations, appear only in a single license. Our approach defines actions, where possible, in terms of rights and obligations defined in U.S. law and the Berne Convention, making ontology building more scalable for large numbers of licenses. The set of tuples chosen for a license and the subsumption relationships between its actions and those of other licenses are determined by the legal interpretation of the license; the remainder of the interpretation, in our view, consists of each action's definition in the world.

The appendix presents one interpretation of the well-known BSD license as tuples, using about a dozen distinct actions and representing about a day's work by one analyst. We find that a license can typically be expressed with a few tens of rights tuples, with each right associated with roughly one to five obligation tuples. The examples of typical rights and obligations listed in the section titled Understanding Open Software Licenses can be interpreted as the following tuples; of course, other interpretations are also possible, and indeed the third provision (from GPLv2) has several prominent ones:

Licensee : may : reproduce <AnySource>

"... each Contributor hereby grants to You a ... copyright license to reproduce ... the Work ... in Source ... form."

Licensor : need-not : remedy liability with respect to <Any>
"In no event shall the authors or copyright holders be liable for any claim, damages or other liability."

Licensee : must : publish [DerivativeWork] under GPLv2
"You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License."
Licensee : must-not : claim endorsement of [DerivativeWork] by {{ORGANIZATION}} or contributors to [Original]

"Neither the name of the <ORGANIZATION> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission."

With this approach, nearly all license provisions can be expressed, specifically, all the enactable, testable rights and obligations. Examples are shown at the beginning of the section Understanding Open Software Licenses and in Figure 2. However, there are certain license provisions that are neither enactable nor testable and thus cannot be expressed in terms of an action. The following are some examples:

An exhortation: "The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users." (source: GPLv2 License)

Non-binding advice: "If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission." (source: LGPLv2.1 License)

An explanation: "For example, a Contributor might include the Program in a commercial product offering, Product X." (source: Common Public License 1.0)

A non-binding request: "It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document." (source: GNU Free Documentation License 1.3)

We argue that non-enactable, non-testable provisions are not relevant to the problem we address, namely, to identify license conflicts and guide architectural design to enable specific licensing results.

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 permissive 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 becomes too cumbersome. Thus, automated support needs to be provided to help overcome and manage the multi-component, multi-license complexity.

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 (Dashofy, Hoek, & Taylor, 2005) to describe OAs that can be designed and analyzed with a software architecture design environment (Medvidovic et al.,1999), such as ArchStudio4 (Dashofy et al., 2007). We have taken this environment and extended it with a Software Architecture License Traceability Analysis module (cf. Asuncion & Taylor, 2012). 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 (Dashofy et al., 2007; Medvidovic et al., 1999).

Figure 3 shows a screenshot of an ArchStudio4 session in which we have modeled the OA seen in Figure 1. OA software components 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.

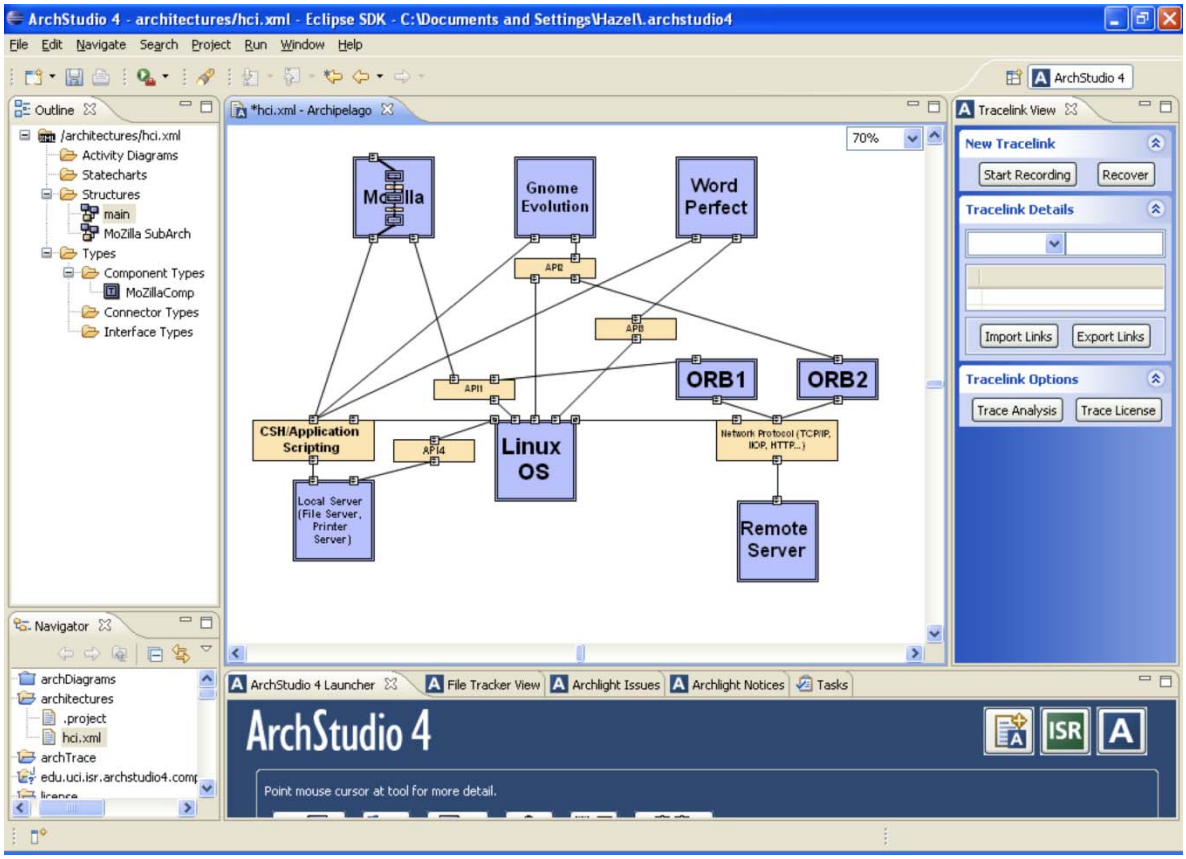

Figure 3. An ArchStudio 4 Model of the Open Software Architecture of Figure 1

Furthermore, the Mozilla component as shown here contains a hypothetical sub-architecture for modeling the role of intra-application scripting, as might be useful in specifying license constraints for Rich Internet Applications. This sub-architecture 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 & Antón, 2008).

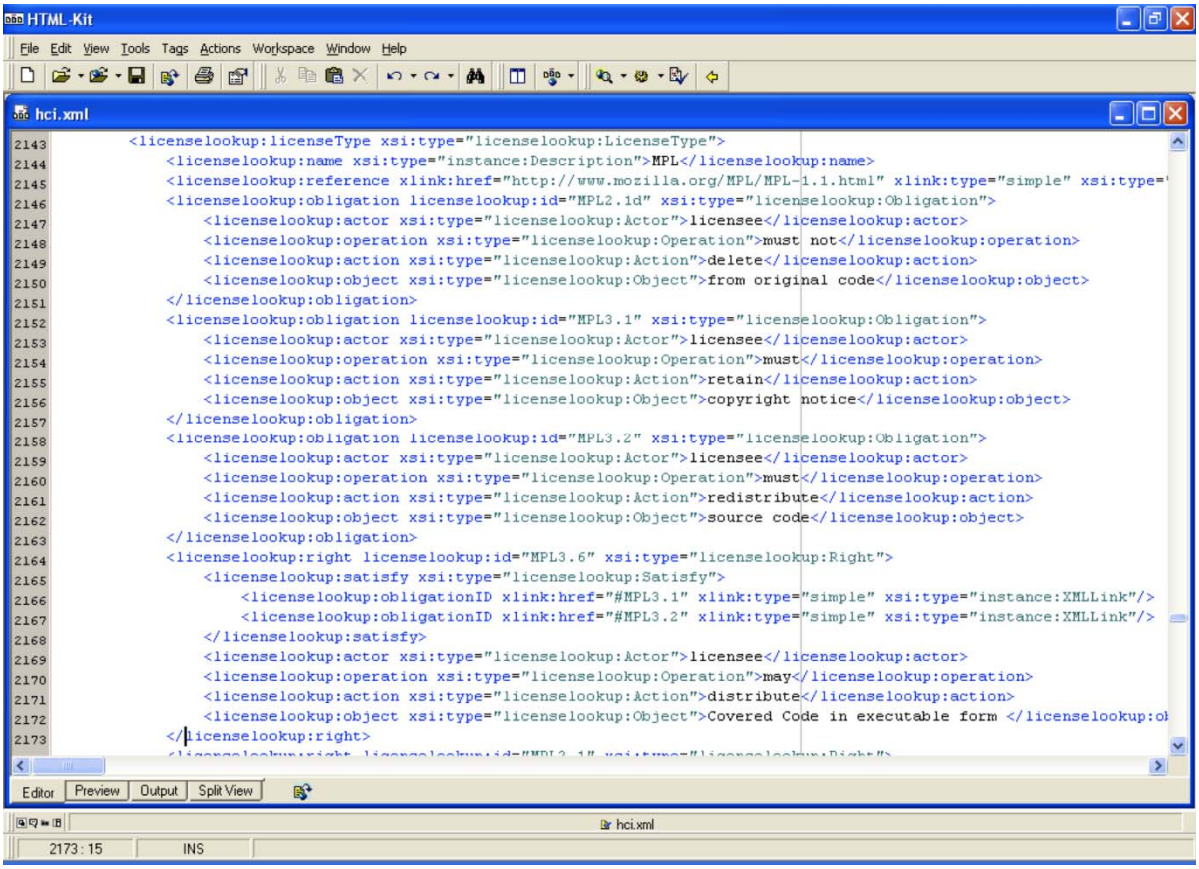

Figure 4. A View of the Internal Schematic Representation of the Mozilla Public License

With this basis to build on, it is now possible to analyze the alignment of rights and obligations for the overall system, in terms of the propagation of reciprocal obligations, licensing conflicts and incompatibilities, and rights and obligation calculations.

- *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 one widely accepted interpretation, namely that build-time static linkage propagates the reciprocal obligations, but that "license firewalls" such as dynamic links or client–server connections do not. Analysis begins, therefore, by propagating these obligations along all connectors that are not license firewalls.

- *Licensing conflicts and incompatibilities*

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, as a specific obligation. 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. Similar conflicts may arise between obligations and desired rights. We discuss this further in the following section.
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 "GUIScriptInterpreter").

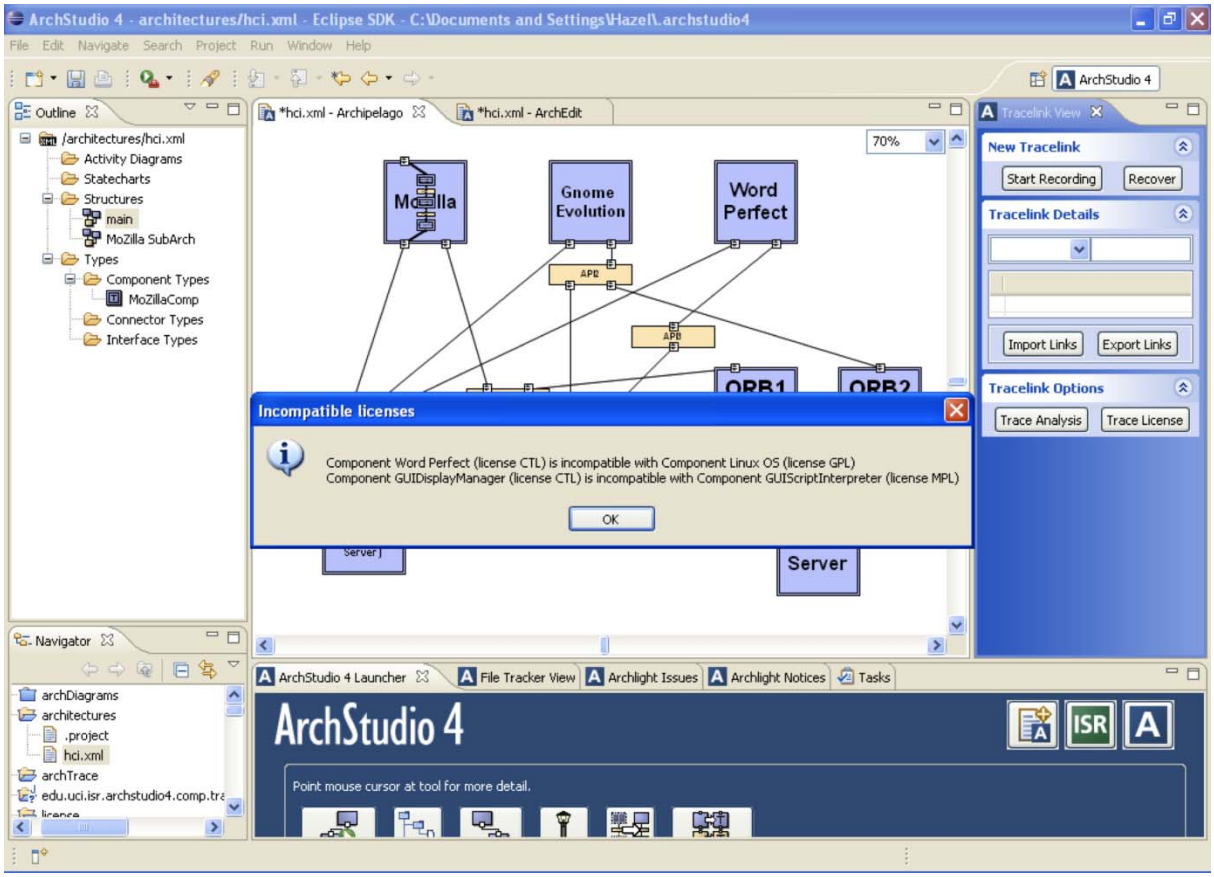

Figure 5. License Conflicts Have Been Identified Between Two Pairs of Components

- *Rights and obligations calculations*

The calculation begins from each right desired for the system as a whole. The right is examined for each component of the system: the right is either freely available (i.e., not an exclusive right defined in copyright or other law), subsumed by some right granted by the component's license, or unavailable. The license tuples for the component are examined for one that subsumes the desired right. If there is no such tuple, then the desired right is unavailable for the component and thus for the system containing it. But

if there is such a tuple, it is instantiated for the component, and the associated obligations are instantiated in parallel. Each instantiated obligation is added to the set of obligations for the system. In addition, the *correlative right* (Hohfeld, 1913) to perform each obligation must be obtained; the calculation recurses with each correlative right as the desired right. The calculation terminates when all chains of rights and obligations have terminated in either a freely available right or an unobtainable right. The result is a set of available instantiated rights, each associated with zero or more instantiated obligations and their correlative rights, and a set (hopefully empty) of unobtainable rights (Alspaugh et al., 2011).

Several kinds of problems may be identified: (1) Desired or correlative rights may be unobtainable; (2) the desired rights may entail obligations that conflict and cannot both be satisfied; (3) desired or correlative rights may be obtainable, but cannot be exercised because they conflict with an obligation; and (4) all desired rights may be available, but the entailed obligations may be more than the system's developers or eventual users are willing to undertake. Examples of specific obligations are as follows:

- OSS: "Licensee must retain copyright notices in the binary form of `module.c`"
- Reciprocal OSS: "Licensee must publish the source code of `component.java` version 1.2.3"
- Proprietary EULA: "Licensee must obtain a proprietary license for a copy of `component.exe`"
- Proprietary ToS: "Licensee must obtain a proprietary license for use of service `http://service.com.`"

Figure 6 shows a report of the calculations for the hypothetical sub-architecture 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 sub-architecture as a whole if the conflict is resolved; a production tool would also list the rights (none) currently available.

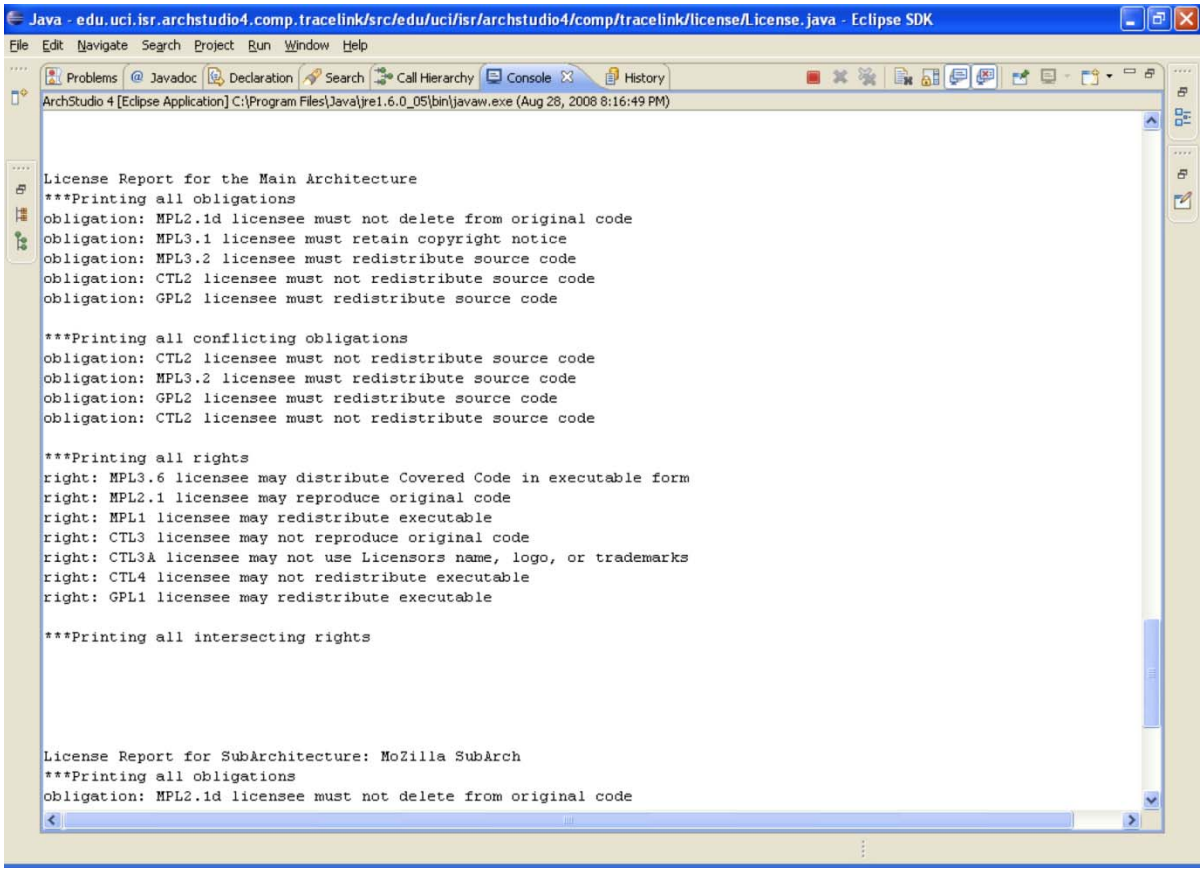

Figure 6. A Report Identifying the Obligations, Conflicts, and Rights for the Architectural Model

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" or "licensor" and seems likely to remain so, and we found use for only four 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 time or run time as it involves the widest range of issues, including representations, rights and obligations calculations, and design guidance derived from them.

We do not claim our approach is a substitute for advice from legal counsel; it is not (and if we claimed it were, such a claim would be illegal in many jurisdictions). The encoding of the BSD license in the appendix is merely an example; we have not developed "the" interpretation, but rather an approach through which many alternative interpretations can be expressed and then worked. Our key contribution is an approach through which inferences can be drawn about licensing issues for a particular design architecture or build- or run-time configuration, based on a particular set of license interpretations. During our research, we have discussed our approach with a number of people in the legal field, including a law professor, a law student working in intellectual property law, an international law researcher, and several lawyers. Our approach implements an inference system based on Hohfeld's (1913) jural relations, which are viewed as foundational in U.S. legal scholarship; follows an inference process accepted by persons with legal training; and uncovers the same kinds of concerns a knowledgeable and thorough analyst would. Our approach provides a way for organizations to express their own interpretations of software licenses, and use those interpretations to rapidly and consistently identify license conflicts, unavailable rights, and unacceptable obligations resulting from a particular architectural configuration. We believe this empowers organizations to steer clear of known problems and highlight issues for analysis by legal counsel.

Based on our analysis approach, it appears that the questions of which license 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 & 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 a 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. A 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.

Solutions and Recommendations

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. An approach for addressing licensing issues at design time such as the one we present here will be increasingly important. 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 et al., 2003; Medvidovic et al., 1999).

Constructing licensing 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.

We note that expressing a software license in our tuples necessarily implies selecting an interpretation of the provisions of the license. Individuals and small organizations may simply choose a representative or commonly accepted interpretation, but enterprises will of necessity seek legal counsel and construct their own interpretations aligned with their advice. An enterprise must also consider the scope of our approach, which focuses on exclusive rights to “do” that are constant over a defined time span, as the copyright rights are. Patent rights, for example, are fundamentally different, being exclusive rights to “prevent from doing” rather than to “do.” For example, the owner of a copyright has the right to copy the work, and may license that right to others who may then make copies. In contrast, the owner of a patent has the right to prevent others from using the

algorithm, process, or invention, and may only grant a license by which the owner will forbear from preventing the licensee from using the patented matter rather than a straightforward right to use it; this license has no effect if some other overlapping patent exists or is granted in the future, and the other patent owner can still prevent the licensee of the first patent from using it. A number of the prominent OSS licenses, such as GPLv3, have provisions for indemnifying licensees against patent infringement involving the licensed material, and our approach supports considering these provisions at system design time.

Advances in the identification and extraction of configured software elements at build time, and their restructuring into architectural descriptions is becoming an endeavor that can be automated (cf. Choi & Scacchi, 1990; Kazman & Carrière, 1999; Jansen, Bosch, & Avgeriou, 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" or "must not" in one and "may" or "need not" 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.

Conclusion

The relationship between enterprise software systems that employ an open architecture, open source software, and multiple software licenses has been and continues to be 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 enterprise strategy for OA systems, 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.

The central problem we examined in this chapter was to identify principles of software architecture and software copyright licenses that facilitate or inhibit how best to ensure

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. Furthermore, in related work, we have gone on to formally model and analyze the alignment, matching, subsuming, and conflicting relationships among interconnected enterprise software components that are subject to different licenses (Alspaugh et al., 2009; Alspaugh et al., 2010).

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, which in our approach is done using an architectural description language. 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-time, build-time, 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.

References

Alspaugh, T. A., & Antón, A. I. (2007). Scenario support for effective requirements. *Information and Software Technology, 50*(3), 198–220.

Alspaugh, T. A., Asuncion, H. A., & Scacchi, W. (2009). Intellectual property rights for heterogeneously licensed systems. In *Proceedings of the 17th International Requirements Engineering Conference (RE '09)* (pp. 24–33). Los Alamitos, CA: IEEE.

Alspaugh, T. A., Asuncion, H. A., & Scacchi, W. (2011). Presenting software license conflicts through argumentation. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE '11).* Knowledge Systems Institute, Skokie, IL.

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

Asuncion, H. (2008). Towards practical software traceability. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08), Companion Volume* (pp. 1023–1026). New York, NY: ACM.

Asuncion, H. U., & Taylor, R. N. (2012). Automated techniques for capturing custom traceability links across heterogeneous artifacts. *Software and Systems Traceability*, Part 2, 129–146. London, UK: Springer-Verlag.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice* (2nd ed.). New York, NY: Addison-Wesley Professional.

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

Choi, S., & Scacchi, W. (1990). Extracting and restructuring the design of large systems. *IEEE Software, 7*(1), 66–71.

Dashofy, E., Asuncion, H., Hendrickson, S. A., Suryanarayana, G., Georgas, J. C., & Taylor, R. N. (2007, May 20–26). ArchStudio 4: An architecture-based meta-modeling environment. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '07), Companion Volume* (pp. 67–68). New York, NY: ACM.

Dashofy, E. M., Hoek, A. v. d., & Taylor, R. N. (2005). A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology, 14*(2), 199–245.

Feldt, K. (2007). *Programming Firefox: Building rich internet applications with XUL.* Sebastopol, CA: O'Reilly Press.

Fielding, R., & Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions Internet Technology, 2*(2), 115–150.

Fontana, R., Kuhn, B. M., Moglen, E., Norwood, M., Ravicher, D. B., Sandler, K., … Williamson, A. (2008). *A legal issues primer for open source and free software projects, version 1.5.1.* Retrieved from http://www.softwarefreedom.org/resources/2008/foss-primer.pdf

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

Jansen, A., Bosch, J., & Avgeriou, P. (2008). Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software, 81*(4), 536–557.

Kazman, R., & Carrière, J. (1999). Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering, 6*(2), 107–138.

Kuhl, F., Weatherly, R., & Dahmann, J. (2000). *Creating computer simulation systems: An introduction to the high level architecture.* Upper Saddle River, NJ: Prentice-Hall PTR.

Medvidovic, N., Rosenblum, D. S., & Taylor, R. N. (1999). A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference Software Engineering (ICSE '99)* (pp. 44–53). New York, NY: ACM.

Meyers, B. C., & Oberndorf, P. (2001). *Managing software acquisition: Open systems and COTS products.* New York, NY: Addison-Wesley.

Nelson, L., & Churchill, E. F. (2006). Repurposing: Techniques for reuse and integration of interactive services. In *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration*., IEEE Computer Society, Los Alamitos, CA.

Open Source Initiative (OSI). (2011). *The Open Source Initiative.* Retrieved from http://www.opensource.org/

Rosen, L. (2005). *Open source licensing: Software freedom and intellectual property law.* Upper Saddle River, NJ: Prentice-Hall PTR. Retrieved from http://www.rosenlaw.com/oslbook.htm

Scacchi, W. (2002, February). Understanding the requirements for developing open source software systems. *IEE Proceedings—Software, 149*(1), 24–39.

Scacchi, W. (2007). Free/open source software development: Recent research results and emerging opportunities. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (pp. 459–468). ACM, New York.

Scacchi, W. (2009). Understanding requirements for open source software. In K. Lyytinen, P. Loucopoulos, J. Mylopoulos, & W. Robinson (Eds.), *Design requirements engineering: A ten-year perspective* (LNBIP 14; 467–494). Springer-Verlag., New York.

Scacchi, W., & Alspaugh, T. A. (2008). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (NPS-AM-08-036; Vol. 1, pp. 230–244). Retrieved from http://www.acquisitionresearch.net

St. Laurent, A. M. (2004). *Understanding open source and free software licensing.* Sebastopol, CA: O'Reilly Press.

Ven, K., & Mannaert, H. (2008). Challenges and strategies in the use of open source software by independent software vendors. *Information and Software Technology, 50*, 991–1002.

Acknowledgments

The research described in this report has been supported by grant #0808783 from the U.S. National Science Foundation, and grants N00244-10-1-0077 and N00244-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School. No endorsement implied.

Key Terms & Definitions

Keywords: Open source software, software licenses, open architectures, closed source software, automated software license analysis, software traceability, interactive development environment.

Open source software: software whose source code is available for external access, study, modification, and redistribution by end-users, accompanied by a software copyright agreement (software license) that ensures these rights are available to anyone who satisfies the explicitly declared obligations included in the licenses. See the Open Source Initiative (OSI, 2011) for other definitions of open source software. In contract, closed source software generally denotes proprietary whose source code is not available for external access, study, modification, or redistribution by end-users, and therefore may also be available as restricted use, executable components.

Software licenses: a contractual agreement conveyed from software developers, owners, or producers to external users of the software, most typically through explicit copyright declarations or an end-user license agreement (EULA).

Open architecture (OA): a software system architecture that explicitly specifies, models, or visually renders the software components of a system, the connectors that interconnect data or control flow between components via each component's

interfaces, that collectively denote the overall architectural configuration of a system in a form that can be accessed, studied, modified, or redistributed.

Architectural description language (ADL): a formal language or notational scheme for explicitly specifying the elements of a software system architecture, including the system's components, component interfaces, and connectors that collectively denote the overall architectural configuration of the system. ADLs are a convenient way to specify an OA software system.

Automated software license analysis: a technique for automatically analyzing the propagation of software copy rights and obligations across interconnected software components that are part of an explicit, open architecture software system.

Software traceability: a technique for navigating or tracing relationships between elements of a software system, and/or documentation of its software engineering.

Architecture development environment : an integrated ensemble of software tools whose collective purpose is to facilitate the interactive development of software system architecture models using an ADL, ideally in a form that can be also used to subsequently develop or consistently derive the build-time and run-time versions of the specified software system architecture.

Appendix: An Interpretation of the BSD 3-Clause License

General obligations for the license as a whole

O1. Licensee : must-not : seek remedy based on warranty or liability with respect to [Any].

Rights and corresponding obligations

R1. Licensee : may : reproduce {AnyOriginalSource}

R2. Licensee : may : reproduce {AnyOriginalBinary}
o2.1. Licensee : must : distribute the BSD notice with [Any]

R3. Licensee : may : prepare derivative work {AnyDerivativeWork} of {AnyOriginalSource}
o3.1. Licensee : must : retain BSD notices from [Original] in [DerivativeWork]
o3.2. Licensee : must-not : claim endorsement for [DerivativeWork] by {{ORGANIZATION}}
o3.3. Licensee : must-not : claim endorsement for [DerivativeWork] by contributors to [Original]

R4. Licensee : may : prepare derivative work {AnyDerivativeWork} of {AnyOriginalBinary}
o4.1. Licensee : must : distribute the BSD notice with [Any]

o4.2. Licensee : must-not : claim endorsements for [DerivativeWork] from {{ORGANIZATION}}
o4.3. Licensee : must-not : claim endorsements for [DerivativeWork] from contributors to [Original]

R5. Licensee : may : distribute copies of {AnyOriginalSource}
o5.1. Licensee : must : retain the BSD notice in [Any]

R6. Licensee : may : distribute copies of {AnyOriginalBinary}
o6.1. Licensee : must : distribute the BSD notice with [Any]

Notes

The BSD license is a template license parameterized by the name of the organization that is the licensor; <ORGANIZATION> in the license text, represented by parameter {{ORGANIZATION}} in the tuple syntax.

BSD grants its own distinct set of rights: redistribution (not distribution, though it is implied) and use, with the right of modification implied but not explicitly granted. This interpretation expresses them in terms of the standard U.S. copyright rights of reproduction, preparation of derivative works, and distribution of copies, similar to the Berne Convention rights.

BSD grants some rights and imposes some obligations that seem superfluous or problematic, and this interpretation provides one rationalization of them. For example, for R1, BSD imposes the obligation to retain the BSD notice, but this right is for the unmodified source, so the obligation has no effect (of course the notice is retained, because the source is unmodified). Another example is the right to use, which is freely available unless the program in question infringes a patent; but BSD contains no provisions for granting a license to infringe any of the licensor's patents and the licensor cannot grant one for other patents, so the grants of the right to use seem superfluous and were ignored in this interpretation.

THIS PAGE INTENTIONALLY LEFT BLANK

The Challenge of Heterogeneously Licensed Systems in Open Architecture Software Ecosystems

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

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. But this may result in possible license conflicts and organizational liability for failure to fulfill license obligations. We have developed an approach for understanding and modeling software licenses, as well as for analyzing conflicts among groups of licenses in realistic system contexts and for guiding the acquisition, integration, or development of systems with open source components in such an environment. This work is based on empirical analysis of representative software licenses and heterogeneously licensed systems, and collaboration with researchers in the legal world. Our approach provides guidance for achieving a "best-of-breed" component strategy while obtaining desired license rights in exchange for acceptable obligations.

Introduction

A substantial number of development organizations are adopting a strategy in which a software-intensive system is developed with an *open architecture* (OA; Oreizy, 2000), whose components may be open source software (OSS) or proprietary with open application programming interfaces (APIs). Such systems evolve not only through the evolution of their individual components, but also through replacement of one component by another, possibly from a different producer or under a different license. With this approach, the organization becomes an integrator of components largely produced elsewhere that are interconnected through open APIs 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 when acquired from a vendor employing a proprietary development scheme.

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,
- open interfaces met by the components,
- degree of coupling in the evolution of related components, and
- rights and obligations resulting from the software licenses under which various components are released, that propagate from producers to consumers.

An example software ecosystem is portrayed in Figure 1.

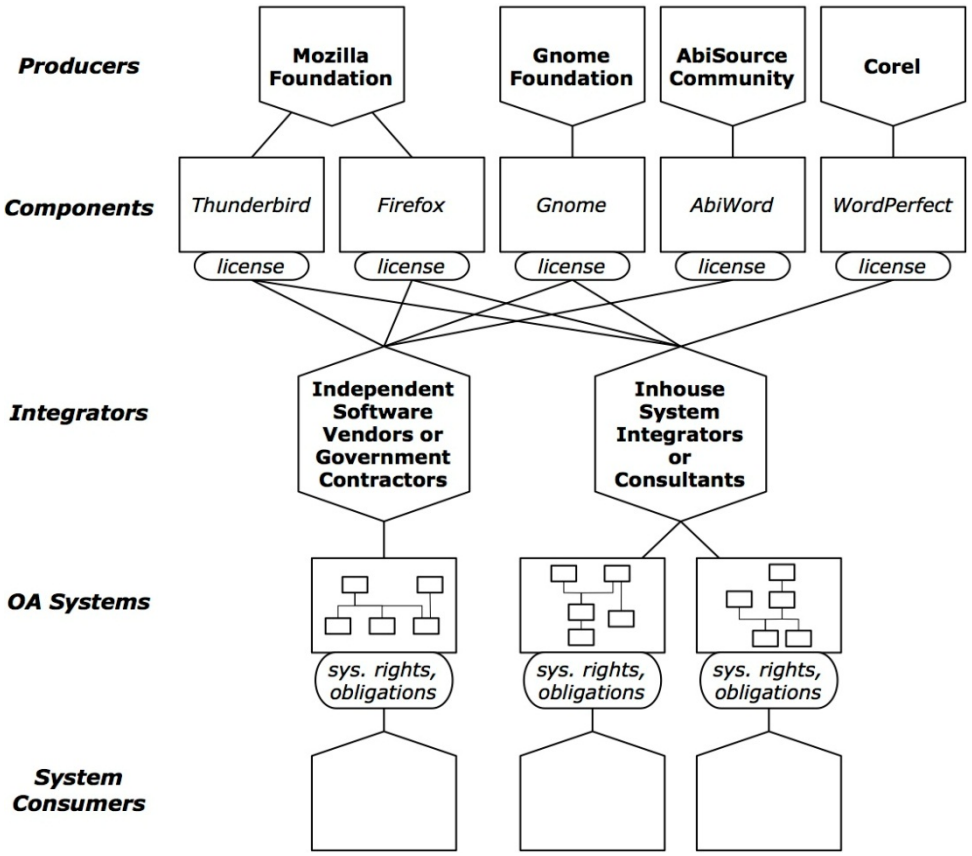

Figure 1. An Example of a Software Supply Network Within a Software Ecosystem in Which OA Systems Are Developed

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 chapter, we build on previous work on heterogeneously licensed systems (Alspaugh, Asuncion, & Scacchi, 2009a; German & Hassan, 2009; Scacchi & Alspaugh, 2008) 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 (2008). Its license agreement, from which we quote in this section, 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, as follows:

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 © 2002–2007 Xiph.org Foundation
8. libtheora, Copyright © 2002–2007 Xiph.org Foundation
9. zlib general purpose compression library, Copyright © 1995–2005 Jean-loup Gailly and Mark Adler
10. libpng PNG reference library
11. jpeglib JPEG library, Copyright © 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 © 1996–2008 Daniel Stenberg <daniel@haxx.se>
16. PostgreSQL Database Management System
17. FreeType, Copyright © 2007 The FreeType Project (www.freetype.org)
18. NVIDIA Cg, Copyright © 2002–2008 NVIDIA Corp.

It is clear that an analysis of the *virtual license* resulting from the interaction of these 18 components and their licenses is unlikely to be straightforward.

An OA system can evolve by a number of distinct mechanisms, as follows, some of which are common to all systems but others of which are a result of heterogeneous component licenses in a single system:

Component evolution—One or more components can evolve, altering the overall system's characteristics (for example, upgrading and replacing the Firefox Web browser from version 3.5 to 3.6).

Component replacement—One or more components may be replaced by others with different behaviors but the same interface, or with a different interface and the addition of shim code to make it match (for example, replacing the AbiWord word processor with either Open Office or MS Word).

Architecture evolution—The OA can evolve, using the same components but in a different configuration, altering the system's characteristics. For example, changing the configuration in which a component is connected can change how its license affects the rights and obligations for the overall system. This could arise when replacing email and word processing applications with Web services like Google Mail and Google Docs.

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.

Changes 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 reciprocal scope of a GPL-licensed module.

The interdependence of integrators and producers results in a co-evolution of software within an OA ecosystem. Closely coupled components from different producers must evolve in parallel in order for each to provide its services, as evolution in one will typically require a matching evolution in the other. Producers may manage their evolution with a loose coordination among releases, for example, as between the Gnome and Mozilla organizations. Each release of a producer component creates a tension through the ecosystem relationships with consumers and their releases of OA systems using those components, as integrators accommodate the choices of available, supported components with their own goals and needs. As discussed in our previous work (Alspaugh et al., 2009a), 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, as follows:

- It must rest on a license structure of rights and obligations, 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 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.
- It must define license architectures.
- It must provide an automated environment for creating and managing license architectures. We have developed a prototype that manages a license architecture as a view of its system architecture (Alspaugh, Asuncion, & Scacchi, 2009b, 2011; Alspaugh, Scacchi, & Asuncion, 2010).
- 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.

In the remainder of this chapter, we survey some related work, provide an overview of OSS licenses and projects, define and examine characteristics of open architectures, introduce a structure for licenses and outline license architectures, and sketch our approach for license analysis. We then close with our conclusions.

Related Work

It has been typical until recently that each software or information system is designed, built, and distributed under the terms of a single proprietary or OSS license, with all its components homogeneously covered by that same license. The system is distributed, with sources or executables bearing copyright and license notices, and the license gives specific rights while imposing corresponding obligations that system consumers (whether external developers or users) must honor, subject to the provisions of contract and commercial law. Consequently, there has been some very interesting study of the choice of OSS license for use in an OSS development project, and its consequences in determining the likely success of such a project.

Brown and Booch (2002) discussed issues that arise in the reuse of OSS components, such as that interdependence (via component interconnection at design time, or linkage at build time or run time) causes functional changes to propagate, and versions of the components evolve asynchronously, giving rise to co-evolution of interrelated code in the OSS-based systems. 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 component version to version.

Legal scholars have examined OSS licenses and how they interact in the legal domain, but not in the context of HLSs (Fontana et al., 2008; Rosen, 2005; St. Laurent, 2004). For example, Rosen (2005) surveyed eight OSS licenses and created two new ones written to professional legal standards. He examined interactions primarily in terms of the general categories of reciprocal and non-reciprocal licenses, rather than in terms of specific licenses. However, common to this legal scholarship is an approach that analyzes the interaction among licenses on a pairwise or interlinked components basis. This analysis scheme means that if system A has an OSS license of type X, system B has a license of type Y, and system C has a license of type Z, then license interaction (matching, subsumption, or conflicting constraints) is determined by how A interacts with B, B with C, and A with C. This follows from related legal scholarship (e.g., Burk, 1998) that brought attention to problems of whether or not intellectual property rights apply depending on how the systems (or components) are interlinked (cf. German & Hassan, 2009). We similarly adopt this approach in our analysis efforts.

Stewart, Ammeter, and Maruping (2006) conducted an empirical study to examine whether license choice is related to OSS project success, finding a positive association following from the selection of business-friendly licenses. Sen, Subramaniam, and Nelson in a series of studies (Sen, 2007; Sen, Subramaniam, & Nelson, 2009; Subramaniam, Sen, & Nelson, 2009) similarly found positive relationships between the choice of an OSS license and the likelihood of both successful OSS development and adoption of corresponding OSS systems within enterprises. These studies direct attention to OSS projects that adopt and identify their development efforts through use of a single OSS license. However, there has been little explicit guidance on how best to develop, deploy, and sustain complex software systems when heterogeneously licensed components are involved, and thus multiple OSS and proprietary licenses may be involved. Ven and Mannaert (2008); Tuunanen, Koskinen, and Kärkkäinen (2009); and German and Hassan (2009) are recent exceptions.

Jansen and colleagues (Jansen, Brinkkemper, & Finkelstein, 2009; Jansen, Finkelstein, & Brinkkemper, 2009) drew attention to their observation that software ecosystems (a) embed software supply networks that span multiple organizations, and (b) are embedded within a network of intersecting or overlapping software ecosystems that span the world of software engineering practice. Scacchi (2007) for example, identified that the world of open source software (OSS) development is a software ecosystem different from those of commercial software producers, and its supply networks are articulated within a network of FOSS development projects. Networks of OSS ecosystems have also begun to appear around very large OSS projects for Web

browsers, Web servers, word processors, and others, as well as related application development environments like NetBeans and Eclipse, and these networks have become part of global information infrastructures (Jensen & Scacchi, 2005).

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

Boucharas, Jansen, and Brinkkemper (2009) then drew attention to the need to more systematically and formally model the contours of software supply networks, ecosystems, and networks of ecosystems. Such a formal modeling base may then help in systematically reasoning about what kinds of relationships or strategies may arise within a software ecosystem. For example, Kuehnel (2008) examined how Microsoft's software ecosystem that emerged for its operating systems (MS Windows) and key applications (e.g., MS Office) may be transforming from "predator" to "prey" in its effort to control the expansion of its markets to accommodate OSS (as the extant prey) that eschew closed source software with proprietary software licenses.

Other previous work examined how best to align acquisition, system requirements, architectures, and OSS components across different software license regimes to achieve the goal of combining OSS with proprietary software that provide open APIs when developing a composite "system of systems" (Scacchi & Alspaugh, 2008). This is particularly an issue for the U.S. federal government in its acquisition of complex software systems subject to Federal Acquisition Regulations (FARs) and military Service-specific regulations. HLSs give rise to new functional and non-functional requirements that further constrain what kinds of systems can be built and deployed, as well as recognizing that acquisition policies can effectively exclude certain OA configurations, while accommodating others, based on how different licensed components may be interconnected.

Open Source Software

Traditional proprietary licenses allow a company to retain control of software it produces, and restrict the access and rights that outsiders can have. OSS licenses, on the other hand, are designed to encourage sharing and reuse of software, and grant access and as many rights as possible. OSS licenses are classified as *permissive* or *reciprocal*. Permissive OSS licenses such as the Berkeley Software Distribution (BSD) license, the Massachusetts Institute of Technology license, the Apache Software License, and the Artistic License, grant nearly all rights to components and their source code, and impose few obligations. Anyone can use the software, create derivative works from it, or include it in proprietary projects. Typical permissive obligations are simply to not remove the copyright and license notices.

Reciprocal OSS licenses take a more active stance towards sharing and reusing software, imposing obligations with respect to the original software in exchange for rights, and also reciprocally on any future derivative versions of it in exchange for the right to create and distribute the derivative versions. The most demanding reciprocal licenses such as GPL impose 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 (OSI, 2008). 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 homogeneously granted and the system has a single well-defined OSS license, was the norm in the early days of OSS and continues to be practiced.

More recently it has become increasingly common for OSS (and mixed) systems to be composed of components from several different organizations governed by different licenses. An example of such a heterogeneously licensed system, this one composed entirely of OSS components, is shown in Figure 2.

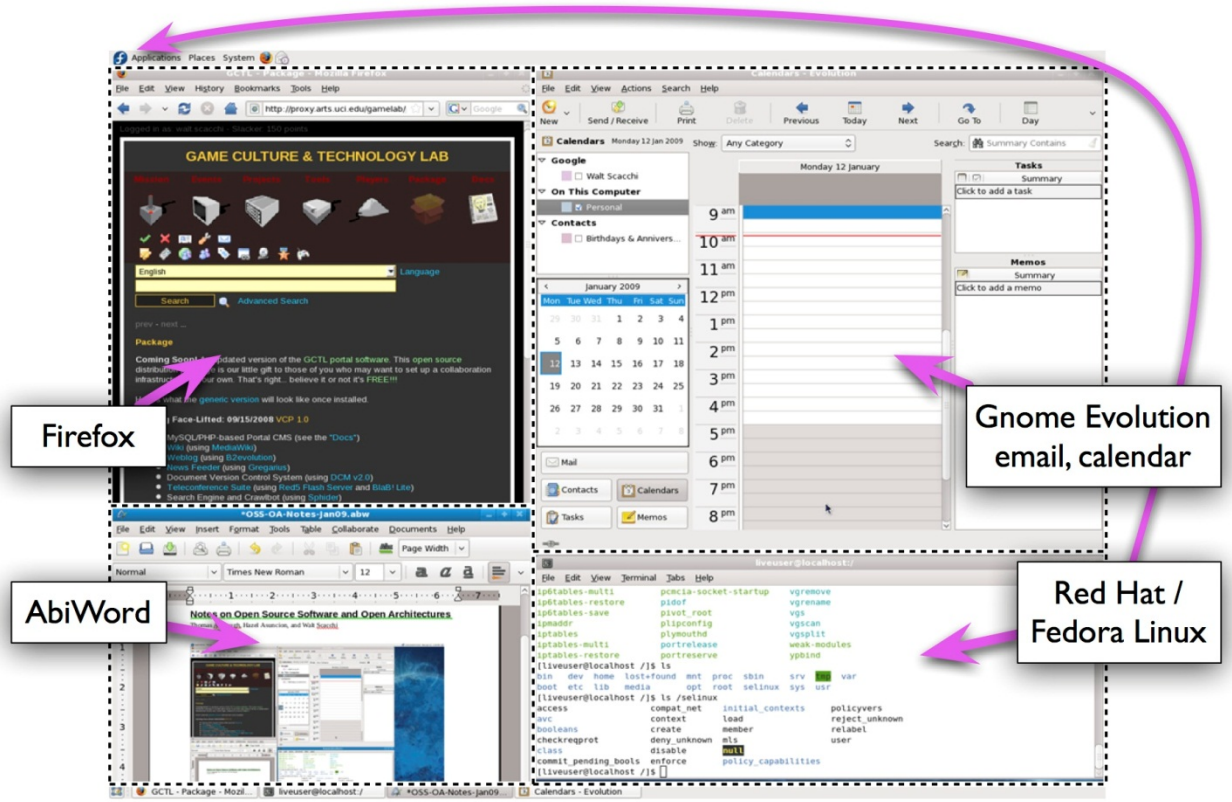

Figure 2. A Heterogeneously Licensed Composite System

Open Architecture

Open architecture (OA) software is a customization technique introduced by Oreizy (2000) that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. Increasingly more software-intensive systems are developed using an OA strategy, not only with OSS components but also proprietary components with open APIs (e.g., see Unity Technologies, 2008). Using this approach can lower development costs and increase reliability and function (Scacchi & Alspaugh, 2008). Composing a system with heterogeneously licensed components, however, increases the likelihood of conflicts, liabilities, and no-rights stemming from incompatible licenses. Thus, in our work we define an OA system as a software system consisting of 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 incorporates OSS components and uses open APIs will result in an OA system. But not all such architectures will produce an OA, since the (possibly empty) set of available license rights for an OA system depends on (a) how and why OSS and open APIs are located within the system architecture; (b) how OSS and open APIs are implemented, embedded, or interconnected; and (c) the degree to which the licenses of different OSS components

encumber all or part of a software system's architecture into which they are integrated (Alspaugh & Antón, 2008; Scacchi & Alspaugh, 2008).

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

Software source code components—These can be either (a) standalone programs; (b) libraries, frameworks, or middleware; (c) inter-application script code such as C shell scripts; or (d) intra-application script code, as for creating Rich Internet Applications using domain-specific languages such as XUL for the Firefox Web browser (Feldt, 2007) or "mashups" (Nelson & Churchill, 2006). 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 (Rosen, 2005). 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" (Meyers & Oberndorf, 2001). Open 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, for example, High Level Architecture (Kuhl, Weatherly, & Dahmann, 1999), CORBA, MS .NET, Enterprise Java Beans, and GNU Lesser General Public License (LGPL) libraries. Connectors can also limit the propagation of license obligations.

Methods of 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 3 shows a high-level view of a reference architecture that includes all the kinds of software elements listed in this section. This reference architecture has been instantiated in a number of configured systems that combine OSS and closed source components. The configured systems consist of software components such as a Mozilla Firefox Web browser, Gnome Evolution email client, and AbiWord word processor (similar to MS Word), all running on a RedHat Fedora Linux operating system accessing file, print, and other remote networked servers such as an Apache Web server. Figure 4 shows a build-time architecture instantiated with those choices. Figure 5 is a screenshot of the instantiated architecture in our extension of ArchStudio (Institute for Software Research, 2006), where it is one view of the architecture data structure whose automated analysis is discussed and shown in later sections. 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.

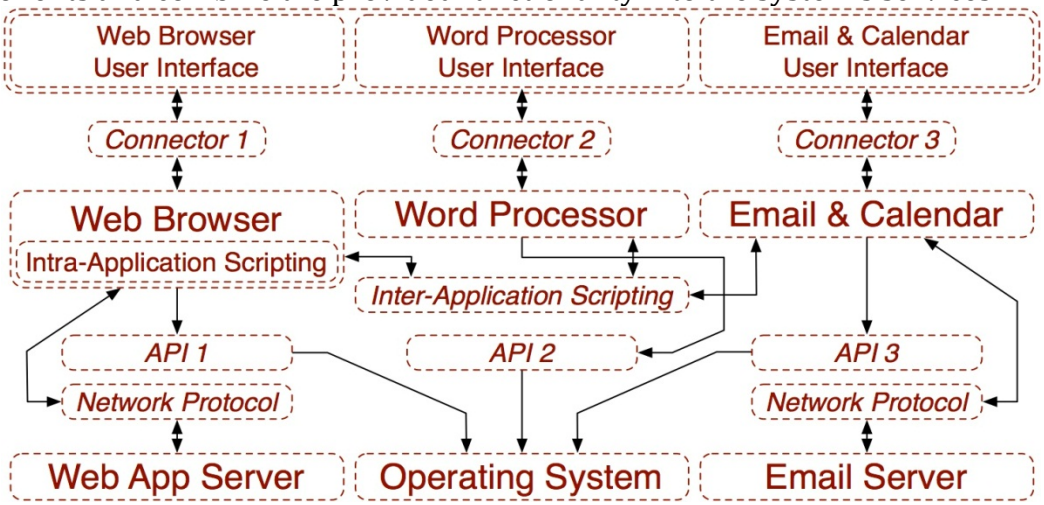

Figure 3. The Design-Time Architecture of the System of Figure 2

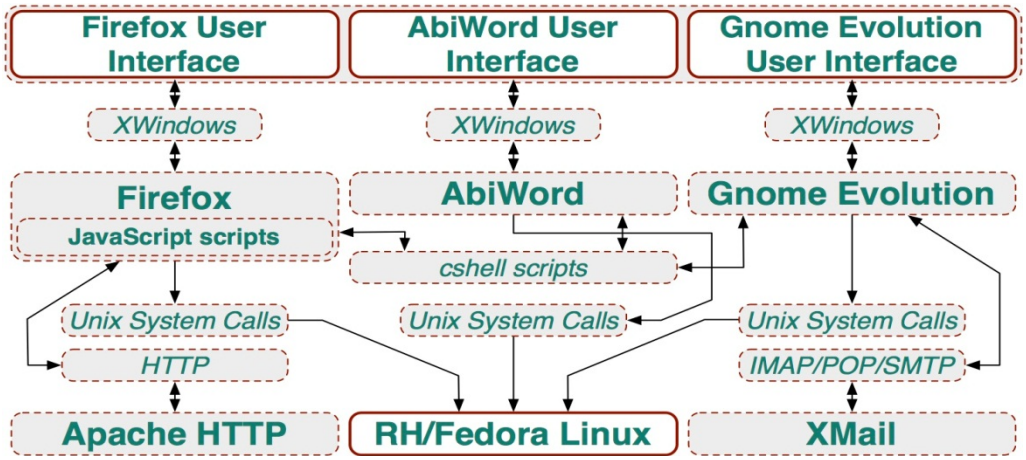

Figure 4. A Build-Time Architecture Describing the Version Running in Figure 2

Note. Components/connectors not visible in Figure 2 are shown in gray.

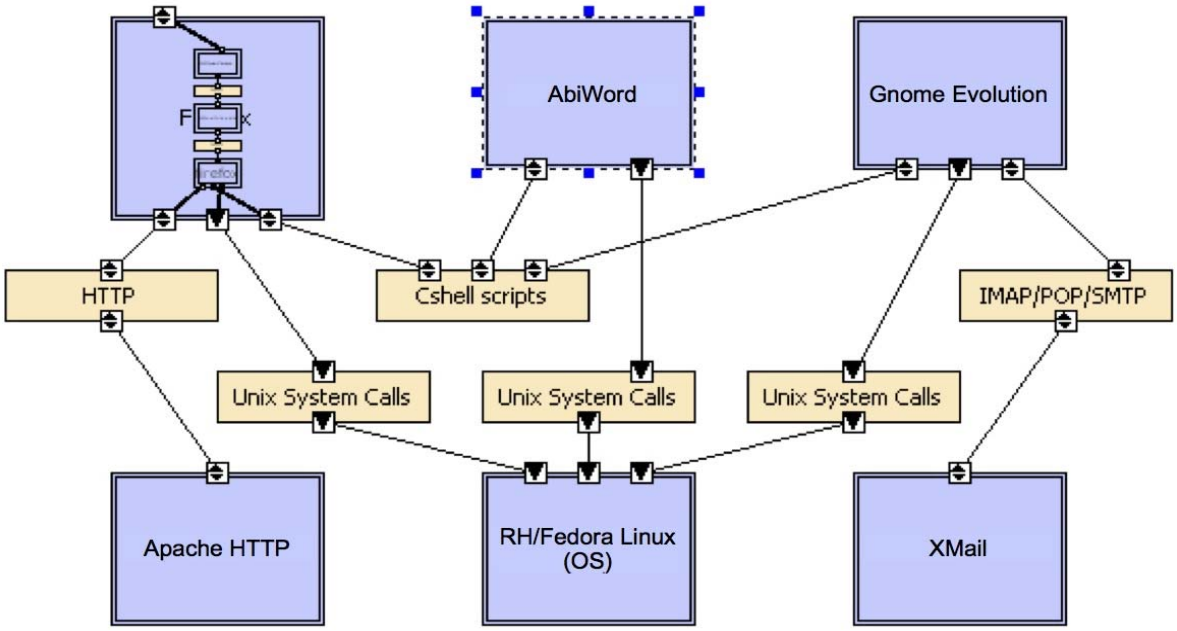

Figure 5. Instantiated Build-Time Architecture (Figure 2) Within ArchStudio

The topology of the build-time architecture also determines the OA software ecosystem of the system, with its dependencies on suppliers and (implicitly) the evolution paths that ecosystem can take, in the context of design and instantiation choices that involve different suppliers of components of the same sort, or more extensive changes that involve suppliers of components of a different sort. Figure 6 shows the reference architecture of Figure 3, annotated with the supplier organizations implied by the instantiations of the build-time architecture of Figure 4. The choices are a result of desired functional abilities and nonfunctional qualities, and should also be influenced by desired supply-chain characteristics, licensing regimes, and future software ecosystem evolution paths. All these choices, however, are limited by software license constraints and interactions, as described in the next two sections.

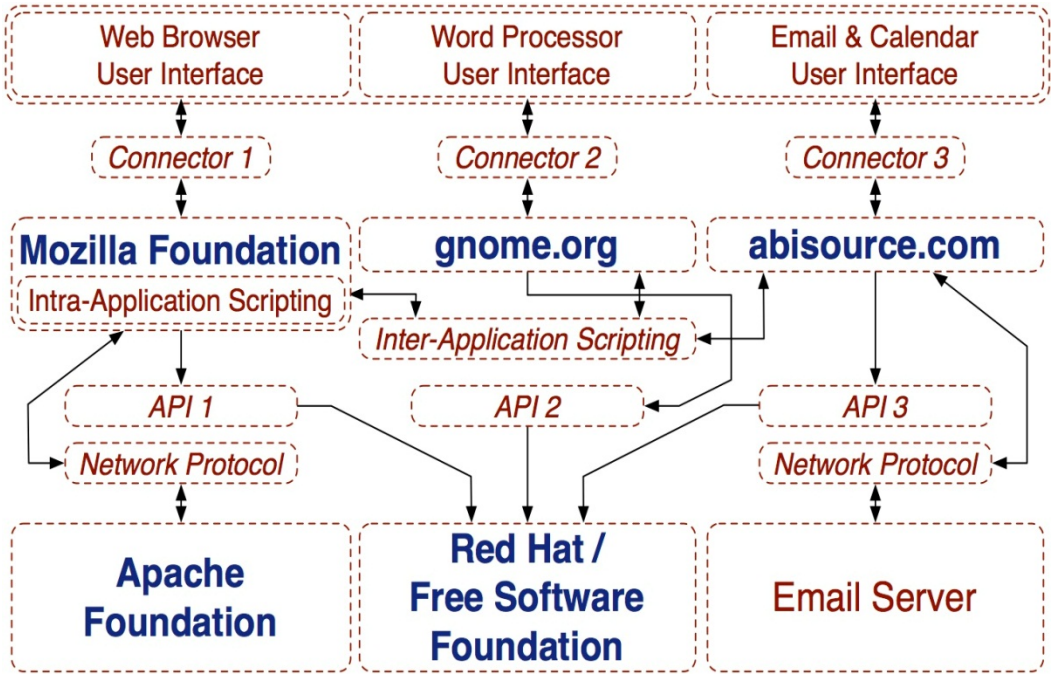

Figure 6. Reference Architecture Components Supplied by an Organization, Indicating the Integrator's Dependencies on Suppliers, Mediated by Interfaces and Licenses

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). The obligations may conflict, as when a GPL'd component's reciprocal obligation to publish source code of other components is combined with a proprietary component's license prohibition of publishing its source code. In this case, no rights may be available for the system as a whole, not even the right of use, because the two obligations cannot simultaneously be met and thus neither component can be used as part of the system.

The basic relationship between software license rights and obligations can be summarized as follows: if the specified obligations are met, then the corresponding rights are granted. For example, if you publish your modified source code and sub-licensed derived works under MPL, then you get all the MPL rights for both the original and the modified code. However, license details are complex, subtle, and difficult to

comprehend and track—it is easy to become confused or make mistakes. The challenge is multiplied when dealing with configured system architectures that compose a large number of components with heterogeneous licenses, so that the need for legal counsel begins to seem inevitable (Fontana et al., 2008; Rosen, 2005).

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 (1913) classic group of eight fundamental jural relations, of which we use *right*, *duty*, *no-right*, and *privilege*. We start with a tuple `<actor, operation, action, object>` for expressing a right or obligation. The actor is the "licensee" for all the licenses we have examined. The operation is one of the following: "may," "must," "must not," or "need not," with "may" and "need not" expressing rights and "must" and "must not" expressing obligations. Because copyright rights are only available to entities who have been granted a sublicense, only the listed rights are available, and the absence of a right means that it is not available. The action is a verb or verb phrase describing what may, must, must not, or need not be done, with the object completing the description. A license may be expressed as a set of rights, with each right associated with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 7 shows the meta-model with which we express licenses, discussed at greater length in our previous work (Alspaugh et al., 2009b, 2011; Alspaugh et al., 2010).

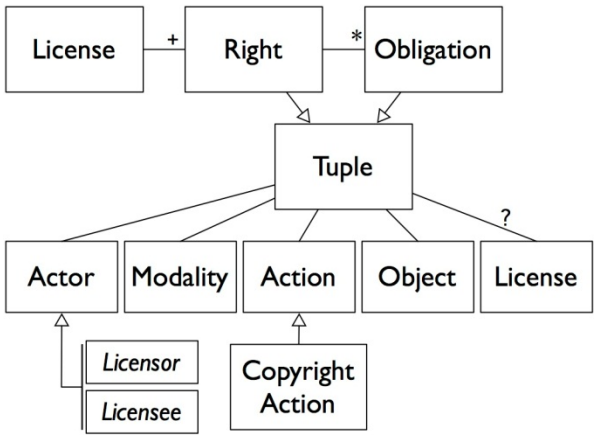

Figure 7. License Meta-Model

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 that includes the:

1. set of components of the system;
2. relation mapping each component to its license (Figure 8);
3. relation mapping each component to its set of sources; and

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

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

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 licensed component through a *license firewall* that limits the scope of reciprocal obligations. Rather than connecting conflicting components directly through static or other build-time links, the connection is made through a dynamic link, client–server protocol, license shim (such as 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 is possible) by, for example, retaining copyright and license notices in the source and publishing the source code. However, even using design heuristics such as these (and there are many), keeping track of license rights and obligations across components that are interconnected in complex OAs quickly becomes too cumbersome. Automated support is needed to manage the multi-component, multi-license complexity.

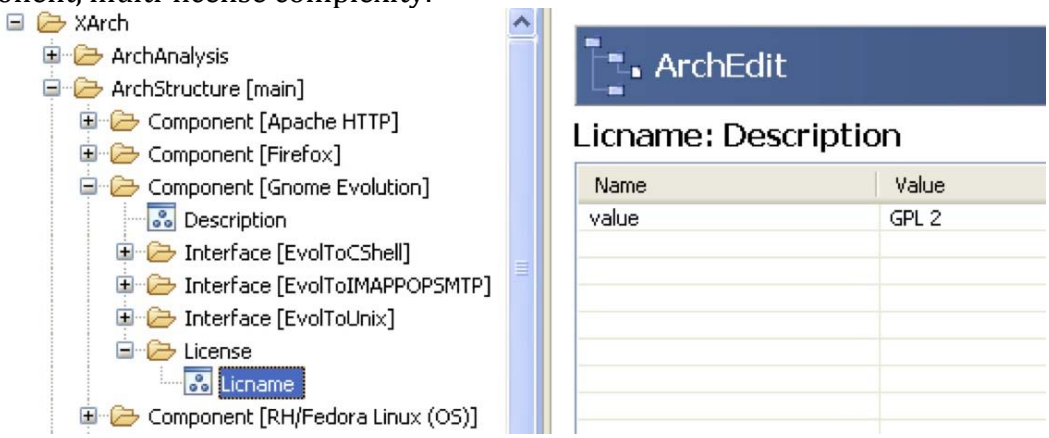

Figure 8. License Annotation of Gnome Evolution Component Seen in Figure 5, as Implemented in our Extension of ArchStudio4
(Alspaugh et al., 2009b, 2011; Alspaugh et al., 2010; Asuncion, 2009)

License Analysis

Given a 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 have developed a prototype of such an environment (Alspaugh et al., 2009b, 2011; Alspaugh et al., 2010).

We use an architectural description language specified in xADL (Institute for Software Research, 2009) to describe OAs that can be designed and analyzed with a software architecture design environment (Medvidovic, Rosenblum, & Taylor, 1999), such as ArchStudio4 (Institute for Software Research, 2006). We have built the Software Architecture License Analysis module on top of ArchStudio's Traceability View (Asuncion & Taylor, 2009). This allows for the specification of licenses as a list of attributes (license tuples) using a form-based user interface in ArchStudio4, shown in Figure 8 (Institute for Software Research, 2006; Medvidovic et al., 1999).
We analyze rights and obligations, as described in the following section (Alspaugh et al., 2009b, 2011; Alspaugh et al., 2010), as implemented in our extension of ArchStudio4 (Alspaugh et al., 2009b; Asuncion, 2009). For example, in Figure 9, we show the results of a license analysis for the architecture shown in Figure 5, showing no conflicts. If we replace AbiWord with Corel WordPerfect, the license analysis then shows what rights are missing, as seen in Figures 9 and 10. The analysis that determines how this works is described in the remaining subsections.

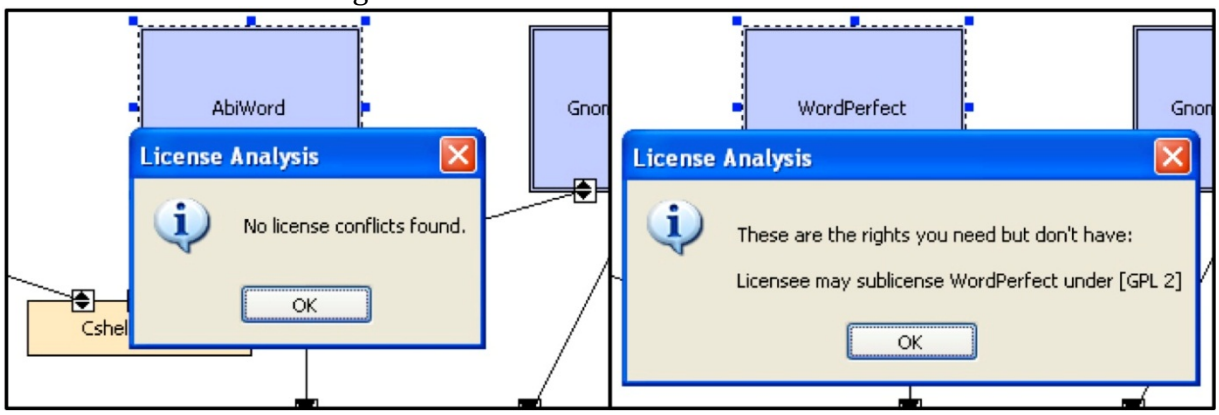

Figure 9. Results of a License Analysis for the Architecture Shown in Figure 5
(Alspaugh et al., 2009b, 2011; Alspaugh et al., 2010; Asuncion, 2009)

Note. On the left is displayed the result of the automated analysis of the architecture in Figure 5. After replacing the AbiWord word processor with the WordPerfect word processor, and redoing the analysis, the tool shows this alternative design results in license conflicts.

Propagation of Reciprocal Obligations

We follow the widely accepted interpretation that build-time static linkage propagates the reciprocal obligations, but appropriate 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 the proprietary license and "must" in GPL.

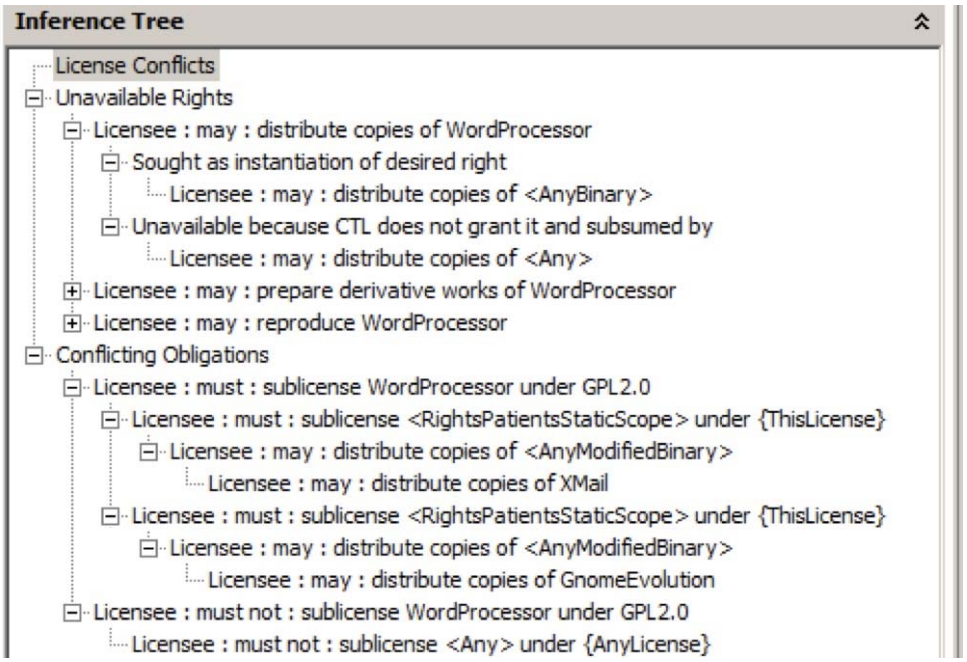

Figure 10. Prototype Explanation Results for a CTL-GPL2.0 Conflict Implemented in our Extension of ArchStudio4
(Alspaugh et al., 2009b, 2011; Alspaugh et al., 2010; Asuncion, 2009)

Note. At the top are the unavailable rights (partially collapsed), and in the middle are two conflicting obligations.

Rights and Obligations Calculations

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 the following 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, …, 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 11 illustrates one step of the application of a license to obtain a desired concrete right *r*. In the license of *r*'s object *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.

Our previous work goes into the calculations in more detail (Alspaugh et al., 2011).

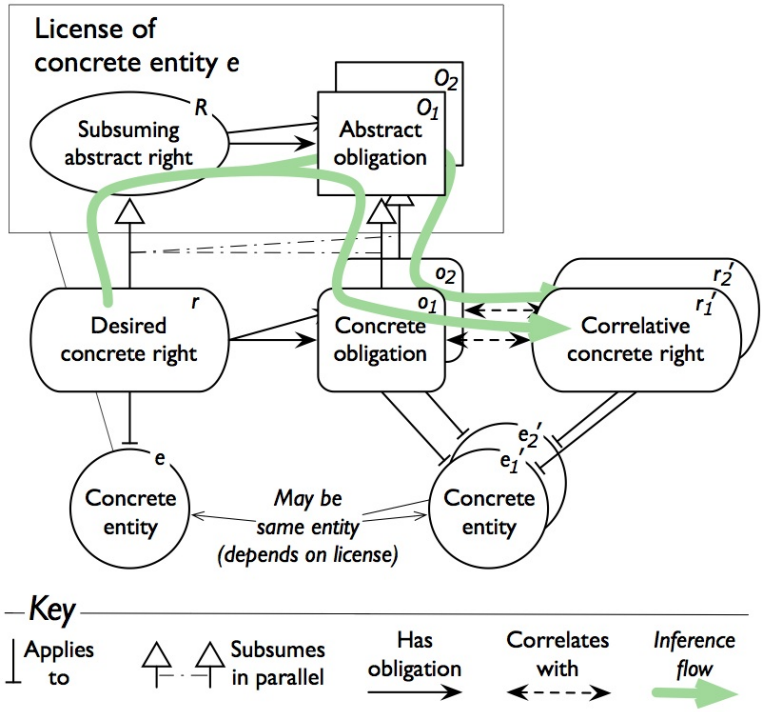

Figure 11. A Step in a Rights/Obligations Inference

Conclusion

This chapter discusses the role of software ecosystems with heterogeneously licensed components 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 software ecosystems with heterogeneously licensed components. 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.

References

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

[2] Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2009a, May). Analyzing software licenses in open architecture software systems. In *Proceedings of the Second*

International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS), IEEE Computer Society, Los Alamitos, CA.

[3] Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2009b, August 31–September 4). Intellectual property rights requirements for heterogeneously-licensed systems. In *Proceedings of the 17th IEEE International Requirements Engineering Conference (RE'09)* (pp. 24–33). Los Alamitos, CA: IEEE.

[4] Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2011, July). Presenting software license conflicts through argumentation. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)* (pp. 509–514). Knowledge Systems Institute, Skokie, IL.

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

[7] Asuncion, H. U. (2009). *Architecture-centric traceability for stakeholders (ACTS)* (doctoral dissertation). Irvine, CA: University of California, Irvine.

[6] Asuncion, H. U., & Taylor, R. N. (2009, May). Capturing custom link semantics among heterogeneous artifacts and tools. In *Proceedings of the Fifth International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)* (pp. 1–5). Washington, DC: IEEE Computer Society.

[8] Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Boston, MA: Addison-Wesley Longman.

[9] Boucharas, V., Jansen, S., & Brinkkemper, S. (2009). Formalizing software ecosystem modeling. In *Proceedings of the First International Workshop on Open Component Ecosystems (IWOCE '09)* (pp. 41–50). New York, NY: ACM.

[10] Brown, A. W., & Booch, G. (2002, April). Reusing open-source software and practices: The impact of open-source on commercial vendors. In *Proceedings of the Seventh International Conference on Software Reuse: Methods, Techniques, and Tools (ICSR-7)*. Springer-Verlag London.

[11] Burk, D. L. (1998). Proprietary rights in hypertext linkages. *Journal of Information, Law and Technology*, 1998(2).

[12] Feldt, K. (2007). *Programming Firefox: Building rich internet applications with XUL*. Sebastopol, CA: O'Reilly Media.

[13] Fontana, R., Kuhn, B. M., Moglen, E., Norwood, M., Ravicher, D. B., Sandler, K., … Williamson, A. (2008). *A legal issues primer for open source and free software projects, version 1.5.1*. Retrieved from http://www.softwarefreedom.org/resources/2008/foss-primer.pdf

[14] German, D. M., & Hassan, A. E. (2009, May). License integration patterns: Dealing with licenses mismatches in component-based development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)* (pp. 188–198). Washington, DC: IEEE Computer Society.

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

[16] Institute for Software Research. (2006). *ArchStudio 4 Web Site.* Retrieved from http://www.isr.uci.edu/projects/archstudio/

[17] Institute for Software Research. (2009). *xADL 2.0* (Technical report). Retrieved from http://www.isr.uci.edu/projects/xarchuci/

[18] Jansen, S., Brinkkemper, S., & Finkelstein, A. (2009). Business network management as a survival strategy: A tale of two software ecosystems. In *Proceedings of the First Workshop on Software Ecosystems* (pp. 34–48). Retrieved from http://www0.cs.ucl.ac.uk/staff/a.finkelstein/papers/twotale.pdf

[19] Jansen, S., Finkelstein, S., & Brinkkemper, A. (2009, May). A sense of community: A research agenda for software ecosystems. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, *Companion Volume* (pp. 187, 190). New York, NY: ACM.

[20] Jensen, C., & Scacchi, W. (2005, July/September). Process modeling across the web information infrastructure. *Software Process: Improvement and Practice*, *10*(3), 255–272.

[21] Kuehnel, A.-K. (2008, June). Microsoft, open source and the software ecosystem: Of predators and prey—The leopard can change its spots. *Information & Communication Technology Law*, *17*(2), 107–124.

[22] Kuhl, F., Weatherly, R., & Dahmann, J. (1999). *Creating computer simulation systems: An introduction to the high level architecture*. Upper Saddle River, NJ: Prentice Hall.

[23] Medvidovic, N., Rosenblum, D. S., & Taylor, R. N. (1999). A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)* (pp. 44–53). New York, NY: ACM.

[24] Meyers, B. C., & Oberndorf, P. (2001). *Managing software acquisition: Open systems and COTS products*. New York, NY: Addison-Wesley Professional.

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

[26] Open Source Initiative. (2008). *The open source definition*. Retrieved from http://opensource.org/docs/osd /

[27] Oreizy, P. (2000). *Open architecture software: A flexible approach to decentralized software evolution* (doctoral dissertation). Irvine, CA: University of California, Irvine.

[28] Rosen, L. (2005). *Open source licensing: Software freedom and intellectual property law*. Upper Saddle River, NJ: Prentice Hall.

[29] Scacchi, W. (2007, September). Free/open source software development. In *Proceedings of the Sixth Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)* (pp. 459–468). ACM, New York.

[30] Scacchi, W., & Alspaugh, T. A. (2008, May). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (Vol. 1, pp. 230–244). Retrieved from http://www.acquisitionresearch.net

[31] Sen, R. (2007). A strategic analysis of competition between open source and proprietary software. *Journal of Management Information Systems*, *24*(1), 233–257.

[32] Sen, R., Subramaniam, C., & Nelson, M. L. (2009). Determinants of the choice of open source software license. *Journal of Management Information Systems*, *25*(3), 207–240.
[33] St. Laurent, A. M. (2004). *Understanding open source and free software licensing*. Sebastopol, CA: O'Reilly Media.
[34] Stewart, K. J., Ammeter, A. P., & Maruping, L. M. (2006). Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects. *Information Systems Research*, *17*(2), 126–144.
[35] Subramaniam, C., Sen, R., & Nelson, M. L. (2009). Determinants of open source software project success: A longitudinal study. *Decision Support Systems*, *46*(2), 576–585.
[36] Tuunanen, T., Koskinen, J., & Kärkkäinen, T. (2009). Automated software license analysis. *Automated Software Engineering*, *16*(3–4), 455–490.
[37] Unity Technologies. (2008, December). End user license agreement. Retrieved from http://unity3d.com/unity/unity-end-user-license-2.x.html
[38] Ven, K., & Mannaert, H. (2008). Challenges and strategies in the use of open source software by independent software vendors. *Information and Software Technology*, *50*(9–10), 991–1002.

Acknowledgments

This research is supported by grants #N00244-10-1-0077 and #N00244-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #0808783 from the U.S. National Science Foundation. No review, approval, or endorsement implied.

THIS PAGE INTENTIONALLY LEFT BLANK

Software Licenses, Coverage, and Subsumption

Walt Scacchi, Thomas Alspaugh, & Rihoko (Inoue) Kawai

Abstract

Software licensing issues for a system design, instantiation, or configuration are often complex and difficult to evaluate, and mistakes can be costly. Automated assistance requires a formal representation of the significant features of the software licenses involved. We present results from an analysis directed toward a formal representation capable of covering an entire license. The key to such a representation is to identify the license's actions, and relate them to the actions for exclusive rights defined in law and to the actions defined in other licenses. Parameterizing each action by the object(s) acted on, the instrumental entities through which the action is performed, and similar contextual variables enables a subsumption relation among the actions. The resulting formalism is lightweight, flexible enough to support the scope of legal interpretations, and extensible to a wide range of software licenses. We discuss the application of our approach to the Lesser General Public License (LGPL) version 2.1.

Introduction

Heterogeneously licensed systems are increasingly prevalent as organizations seek lower development costs, increased reliability and quality, and faster development cycles (Alspaugh, Asuncion, & Scacchi, 2009a; Alspaugh, Scacchi, & Asuncion, 2010; German & Hassan, 2009). Such systems present challenges in ensuring all pertinent obligations from the various possibly conflicting licenses are met, which can easily involve evaluating dozens of distinct licenses applied on a component-by-component basis (Alspaugh, Asuncion, & Scacchi, 2009b; Gobeille, 2008). The challenges arise independently during design, development, integration, distribution, configuration, and execution, and may present different concerns involving different fundamental copyright and patent rights at each of these stages. The goal of our research is to provide licensing guidance to designers, developers, system integrators, and those responsible for software acquisition.

In our previous work we presented an approach and proof of concept of automated licensing analysis integrated into system design at the level of software architecture (Alspaugh et al., 2009a, 2011; Alspaugh et al., 2010). The work was based on a sequence of grounded-theory analyses on (eventually) 46 software licenses, focusing on propagation of obligations through the architectural configuration (the most challenging area for manual analysis). Rights and obligations were the fundamental units of the meta-model we obtained for licenses, with actions as components of rights and obligations and implicitly parameterized in a fixed pattern accommodating the license provisions that were the focus of the work. The textual analyses aimed for coverage of key provisions across a wide range of licenses.

The present work, in contrast, focuses on complete coverage of all license provisions, especially those that might not fit the earlier meta-model. License provisions are represented using a more flexibly extensible approach in which the fundamental unit is an action. Actions are parameterized as needed, recognizing that the subsumption relationship that can be inferred among actions is determined by the form in which the actions are parameterized. Rights and obligations then express relationships among desired, required, and forbidden actions. During our analysis we identified subsumption relationships among actions, linking each action involved in a license right back to the exclusive right subsuming it defined in copyright and other intellectual property law, specifically the U.S. Copyright Act (1976) and the Berne convention (Berne Convention, 1979). Where possible, we also identified subsumptions of the actions of license obligations by the actions of rights. Figure 1 shows the subsumption relationships identified for a single license's actions.

In this paper we focus on the Lesser General Public License (LGPL), version 2.1 (Free Software Foundation, 1999). LGPLv2.1 is the seventh most widely used open-source software (OSS) license, accounting for about 6.5% of open source projects (Black Duck Software, n.d.). At 4,341 words, it is substantial (almost double the mean length of licenses we have analyzed), yet small enough to be discussed manageably. It addresses the most challenging license interaction issue, propagation of obligations to components under other licenses, in a relatively straightforward way compared to other licenses that do so. It has provisions in many of the categories that are challenging for analysis, including the following:

- accumulation of copyright notices,
- alternative obligations,
- clauses with null effect,
- definitional clauses,
- the distinction between collective and derivative work,
- distribution under alternative licenses,
- distinct rights and obligations for build scripts, interfaces, header files, source, object, and executable forms, + license acceptance and termination,
- license exceptions,
- license notices of several types,
- output from licensed software, and
- relicensing under other licenses.

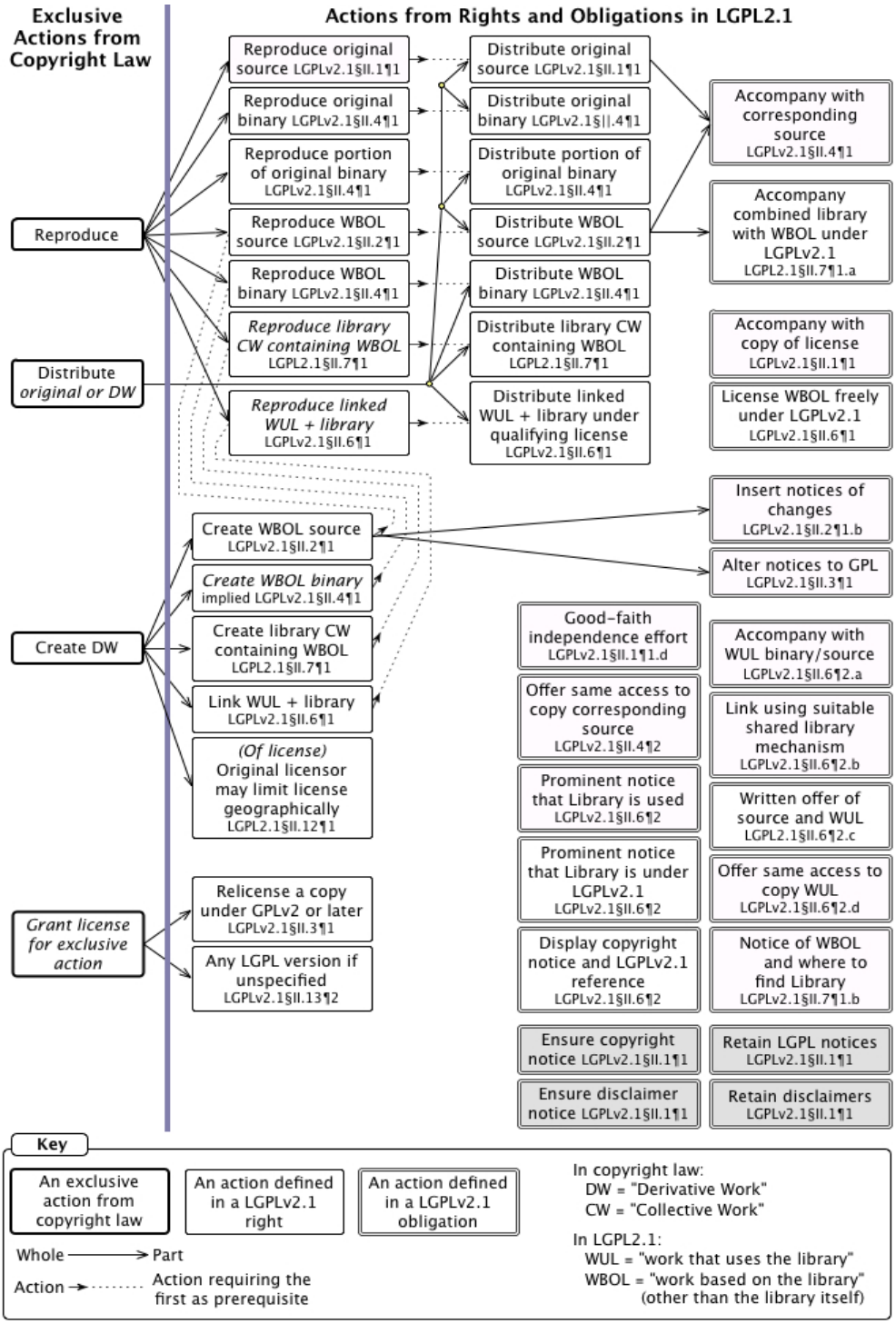

Figure 1. Subsumption Among the LGPLv2.1 Actions for Rights and Obligations and the Exclusive Copyright Actions

Note. Eighteen rights actions are explicit in the text, and three others are implied. The actions of the implied rights are italicized in the figure. Four obligations actions have no effect under the conditions in which they are obligated (because the original source must itself satisfy LGPLv2.1); they are shown with a gray background.

Figure 2 shows an excerpt of the open-coded LGPLv2.1 text annotated with some of the 93 categories that were identified here or in other licenses. The entire license text was chunked and open-coded, reiterating until the boundaries of chunks of text and the conceptual code characterizing each chunk of text stabilized. The list of codes (or categories) was initialized with the codes obtained from our previous analyses of many licenses, and extended to include the kinds of features uncovered by a focused analysis of the LGPLv2.1 text. Portions of the chunking and open-coding were verified by one of the other authors at several points in the process. Axial coding was then used to identify themes and relationships in the license text, resulting, for example, in the categories of definitions, rights, obligations, modifiers, and null effect discussed in the Textual Analysis section, and the characterization of a parameterized action as the basic unit of software licenses discussed in a later section.

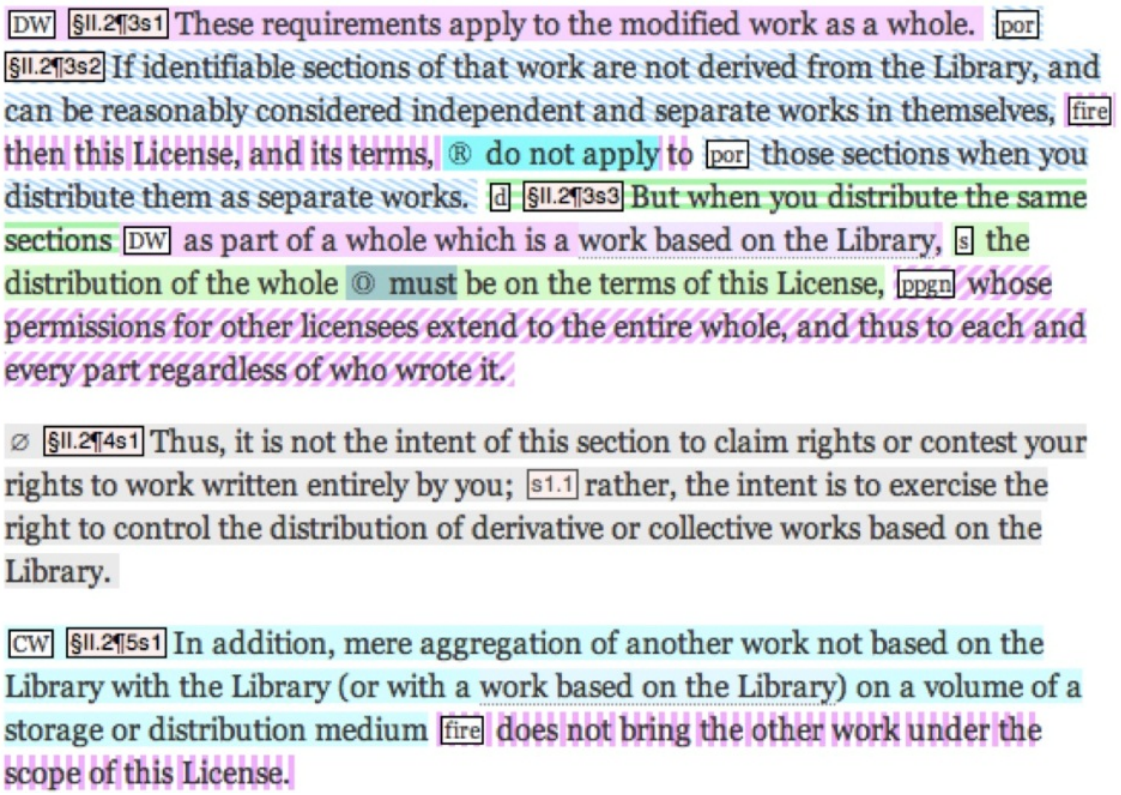

Figure 2. A Portion of LGPLv2.1, Divided Into Chunks and Annotated With Categories of Interest (Alspaugh 2011).

Note. The categories appearing here are, briefly: CW: collective work; d: distribution; DW: derivative work; fire: license firewall; O: apparent obligation; por: for a portion of the licensed entity; ppgn: propagation of obligations to other entities; s: sublicensing, whether explicit or in effect; and ∅: null effect.

LGPLv2.1, like most licenses, is only partially organized into numbered sections, hampering reference to specific parts of the text. Citations of specific license sections, paragraphs, and sentences refer to an online copy of LGPLv2.1 consistently numbered throughout by a program (Alspaugh, 1999).

The remainder of this paper is organized as follows. In the next section, we present related work. After that, we list questions of interest that guide this work. Next is the textual analysis on which the work is based. In the sections that come after, we discuss actions, sketch how actions are parameterized, and outline subsumption among parameterized actions. The final sections include a discussion of some issues arising from the work, followed by the conclusion of the paper.

Related 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").

There has been much work on analysis of laws in Artificial Intelligence (AI) over the past few decades. A widely cited example is Sergot et al.'s (1986) re-expression of the British Nationality Act as a Prolog program; the resulting program was able to apply the Act to a particular person's situation and characteristics to determine nationality. Sergot asserted that the primary value of their approach was the insight that the process of expressing a statute gave into what the statute says and means, rather than any use of the Prolog program.

Otto and Antón (2007) surveyed the literature on modeling legal texts and reached similar conclusions. They highlighted the possibilities of conflicts among regulations, the evolution of case law and passage of new regulations, and the frequent cross-references within a single text or from one text to other texts. They surveyed a number of modeling approaches, including symbolic logic, knowledge representation (including Sergot et al. [1986]), deontic logic, defeasible logic, temporal logic, and so forth.

None of these approaches appear well suited to the challenges of licenses. The problem of references among documents, prominent in the modeling of statutes and regulations, is not significant for the licenses we have examined, which make few references to other documents and exhibit comparatively straightforward references within the license. While the modeling approaches offer a certain degree of automatic calculation, the calculations they support well do not appear particularly useful for OSS licenses. The key issue we have found for OSS licenses, namely how license provisions refer to specific entities in the licensed system and how obligations resulting from rights for one entity are propagated to other entities based on the architectural structure connecting them, is unlike anything addressed by these approaches. While they may possibly offer an efficient run-time implementation for the calculations needed for licenses, it is not clear that they are particularly appropriate for modeling licenses.

Questions of Interest

Our work is in the context of software development and the issues and concerns that arise there. The primary goal in that context is to produce a software system for which the desired rights are available in exchange for acceptable obligations. Since the work originated in the context of the OSS community, we assume goodwill on the part of the actors involved, and do not concern our work with approaches for subverting license provisions. We list the following questions of interest that guide our research:

1. What rights are potentially available for a single component under a given license, or for a given architectural configuration of components and connectors under their licenses?
2. What obligations must be fulfilled in order for specific rights to be granted for a given single component under its license, or for a given architectural configuration of components and connectors under their licenses?
3. What license conflicts (if any) arise for a given architectural configuration of components and connectors under their licenses?

We find that in addressing these questions, we need not consider any license provision that is neither *enactable* nor *testable*. A surprising number of license provisions fall into these categories, including text that we characterize as exhortations, examples or informal explanations of other clauses, and hopes on the part of the licensor that have no legal force. Our focus on testable provisions also leads in the intriguing direction of automated verification of whether a testable license obligation has been fulfilled.

Textual Analysis

We find that everything in the text of LGPLv2.1 (Free Software Foundation, 1999) may be classified as either:

- the definition of a term,
- a right,
- an obligation,
- a modifier to a definition, right, or obligation, or
- text without legal effect.

These five categories cover the entire text and partition everything in it. Examples of each from LGPLv2.1 (Free Software Foundation, 1999) are given below for readers unfamiliar with OSS licenses.

Definitions of Terms

The first example is an explicit definition of a named term, “work that uses the Library.”

> A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library.” (Free Software Foundation, 1999, § II.0¶2)

The second example is an implicit definition of an anonymous category of executables that might be termed "work using the Library and linked with it." Executables in this category have rights and obligations different from those for other executables. LGPLv2.1 gives this category no name.

> … you may also combine or link a work that uses the Library with the Library to produce a work containing portions of the Library … (§ II.6¶1s1)

Rights

The first example, as is common for statements of rights in OSS licenses, grants several rights at once (the right to copy and the right to distribute). The actions in this right might be summarized as "reproduce complete original" and "distribute complete original." We use such summaries here as tokens representing the full definitions.

> You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium … (§ II.1¶1s1)

The second example grants an interesting right to license a specific copy of a work received under LGPLv2.1 under another license. In both these examples, the word "may" signals that a right is probably being defined. The action might be summarized as "license a given copy under GPL."

> You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. (§ II.3¶1s1)

Obligations

The first example is signaled by the word "provided," unlike most LGPLv2.1 obligations which are signaled by "must" (Free Software Foundation 1999). This obligation is notable because it would seem to require no action unless the original source code, in violation of LGPLv2.1, failed to include such a notice and disclaimer of warranty. The actions might be summarized as "ensure appropriate copyright notice" and "ensure disclaimer of warranty."

> … provided that you conspicuously and appropriately publish on each copy (of the complete original source code) an appropriate copyright notice and disclaimer of warranty … (§ II.1¶1s1)

The second example contains no such identifying words, but is the first of a list of alternatives preceded by "… you must do one of these things." Its action might be summarized as "accompany with corresponding source." Many OSS licenses contain similar obligations.

> Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work … (§ II.6¶2.as1)

Modifiers

This first example contains the signal word "provided" that often indicates an obligation, but it does not function as such. Instead its effect is to restrict what "terms of your choice" refers to.

> … you may also combine or link a work that uses the Library with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications. (§ II.6¶1s1)

The second example limits the scope of the anonymous category of "works that use the Library" that are also "works based on the Library" because they incorporate material from header files.

> If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted … (§ II.5¶4s1)

Text Without Legal Effect

The first example that follows is an explanation and statement of the intent of the license's authors; we are told that if the explanation differs from what it purports to explain, their stated intent would be trumped by what the license actually says.

> Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you … (§ II.2¶4)

The second example is more problematic. It is phrased as an obligation, but the action involved ("make a good faith effort") is in our view not testable; compare for example the undoubtedly testable action "conspicuously and appropriately publish on each copy an appropriate copyright notice" (§ II.1¶1s1). Of course, a specific legal interpretation of LGPLv2.1 might give this text a testable interpretation, for example, by operationalizing "good faith effort" in some way.

> If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful. (§ II.2¶1.d)

Other Features

In addition to the five categories of definitions, rights, obligations, modifiers, and null effect that jointly partition and cover the entire LGPLv2.1 (Free Software Foundation, 1999) text, we identified or confirmed several other significant license features.

1. *Right/obligation structure:* All rights and obligations shared the conceptual structure of an actor, a Hohfeld jural relation, and an action. The actor was the licensee for each of LGPLv2.1's 18 rights and 20 obligations. The jural relation was that of a Hohfeld right ("may") or privilege ("need-not") for each right, and of a duty ("must") or no-right ("cannot") for each obligation. Actions are discussed in the section that follows.
2. *Time and state:* Time and state are barely present in LGPLv2.1, figuring only in license acceptance (§ II.9) and termination (§ II.8). There is no provision for reinstatement after termination.
3. *Obligation propagation:* Propagation of obligations to other entities is mediated structurally by the architecture in which LGPLv2.1-licensed entities are combined
    a) to other elements incorporated into the same library (§ II.2¶1.c);
    b) to programs designed to use an LGPLv2.1-licensed library, when linked to the library (§ II.5¶2), except if certain obligations are met (§ II.6¶1); and
    c) to the object code for modules that include more than a stated amount from an LGPLv2.1-licensed header file (§ II.5¶3).
4. *Enactability and testability:* The constructs that appear intended as LGPLv2.1 rights or obligations all involve actions that are clearly testable, with the single exception of the "good faith effort" obligation discussed previously. Every action (even the questionable one) is, unsurprisingly, enactable.

Actions, the Central Construct

Actions are the most common constructs in LGPLv2.1 (Free Software Foundation, 1999), and are essential in how the license is applied in the world. Focusing on actions as the key element of licenses brings several advantages, as follows:

- Actions are more manageable than rights and obligations. Each action is a concept representing an unbounded set of instances of the action; for example, "distribute the Library … in object code … form" (§ II.4¶1) is instantiated by "distribute MONOSPACE `glibc` to John Doe on 2012 June 18" along with any number of similar instances. Therefore, set operations may be used on actions. The operations on rights and obligations, in contrast, are quite limited. For example, the common idiom of first stating an obligation to do action X, then reducing it by granting the right to not do W where W overlaps with or is part of X, is easily expressed as set subtraction

on the actions ITALIC(X-W) but has no simple expression in terms of the obligation and right themselves.

- A single action, or two actions related by subsumption, often appear in both a right and an obligation. In LGPLv2.1, examples are numerous, for example, the obligation "You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change" (§ II.2¶1.b) whose action is subsumed by that of the right "You may modify your copy or copies of the Library" (§ II.2¶1). This phenomenon is essential to the propagation of obligations from one license to entities under another license, which doesn't work unless the other license permits the actions required by the propagated obligations.
- Distinguishing an actual right or obligation from a modifier in the form of a right or obligation can be problematic, as observed in Section IV, but in our analysis we found identifying actions to be uniformly straightforward.
- If rights and obligations are the primary constructs, then their similarities (both comprise an actor, a Hohfeld jural relation, and an action) and unwieldy difference (though each contains a Hohfeld relation, it can't be the same one) are prominent and difficult to justify. But if actions are the primary construct, then rights and obligations become emergent phenomena arising from the relationships among a license's desired, required, and forbidden actions, and the description of the license meta-model becomes simpler and more uniform.

Action Parameterization

In our previous work, we proposed that actions be parameterized with the entity on which they acted, if any, and the license used in the action, if any. However, a more careful examination of license rights and obligations showed that this simple pattern, while sufficient for the majority of current rights and obligations, is neither universally sufficient nor conceptually necessary. Our current work has shown that no fixed pattern or patterns is necessary for formalization and automated inference, and that a small but stubborn set of actions cannot be accommodated in that way. LGPLv2.1 (Free Software Foundation, 1999) offers two examples, of which the following (from a right) is the clearest.

In the action "distribute that work under terms of your choice" (§ II.6¶1), the work in question is a "work that uses the Library" combined or linked with the Library, and the terms in question must meet two conditions (licensee may modify the work, and may reverse-engineer the work). This action thus involves two entities

1. "that work," upon which the action is taken, and
2. the "terms of your choice" through which the distribution is licensed.

Each of these should be a separate parameter of the action, because they vary from instance to instance of the action and may vary independently of each other. If the

action is parameterized in this way, then it becomes a special case of the general action "distribute an entity under a license" and its parameters place it properly in the subsumption hierarchy, as discussed in the following section on Parameterized Subsumption.

(We note in passing that the approach in our previous work for parameterizing the actions of obligations through functions or quantifiers operating on the parameter(s) of the corresponding right continues to suffice, based on the results reported here; we refer interested readers to that work [Alspaugh et al., 2009a, 2011; Alspaugh et al., 2010].)

Parameterized Subsumption

In addressing subsumption among parameterized actions, we follow the approach of Abadi and Cardelli (1996) in the area of object-oriented type systems . Figure 3 illustrates subsumption between pairs of simple actions and pairs of parameterized actions.

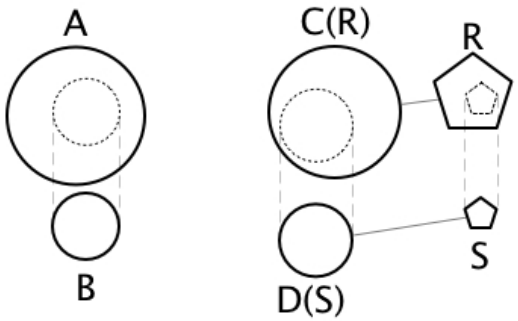

Figure 3. Subsumption of Simple Entities and Parameterized Entities

In the figure, every instance of action *B* is also an instance of action *A*; we say *A* subsumes *B*.

On the right is a more complex situation. Actions *C*(*P*) and *D*(*P*) are parameterized with arguments *R* and *S,* respectively. As is normally the case for parameterized actions in licenses, the parameter is *covariant*: the sense of the subsumption of the arguments matches their effect on the subsumption of the actions they parameterize. Every instance of *D*(*S*) is an instance of *C*(*R*) if every instance of *S* is an instance of *R*; argument *S* is subsumed by argument *R,* so therefore, *D*(*S*) is subsumed by *C*(*R*).

An example from LGPLv2.1 (Free Software Foundation, 1999) is the right "You may modify your copy or copies of the Library" (§ II.2¶1) and the obligation "You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change" (§ II.2¶1.b). In other licenses, we have seen actions to modify licensed entities other than libraries, and to insert various kinds of notices appropriate for the license in question, so we propose generalizing these actions to covariantly parameterized actions informally defined as

M(F, L) = "modify source file F licensed under L"

N(F, L) = "add change notices appropriate for license L to source file F licensed under L"

Let us assert that *M* subsumes *N* covariantly

$$N(g, m) \subseteq M(f, l) \text{ if } g \subseteq f \text{ and } m \subseteq l$$

and also assert that

> "modify your copy or copies of the Library" (Free Software Foundation, 1999, § II.2¶1)

is equivalent to the union of *M*(*F*, `LGPLv2.1`) for each Library file *F* you modify, and that

> "cause the files modified to carry prominent notices stating that you changedthe files and the date of any change" (§ II.2¶1.b)

is equivalent to the union of *N*(*F*, `LGPLv2.1`) for each Library file *F* you modified. Then we have taken several steps towards being able to automatically determine that modifying `libfile.c` under LGPLv2.1 subsumes adding LGPLv2.1 changes notices to `libfile.c`. Our assertions have expressed part of the interpretation of the two actions, and constituted a step in the formalization of an interpretation of LGPLv2.1 as a whole.

We have made such formalizations of portions of licenses in our previous work, and believe the present work provides a foundation for doing the same for entire licenses.

Discussion

In this section, we discuss some of the issues arising from this approach.

What Is and Is Not Formalized?

The following are formalized:

- the subsumption relation among actions, including the effect of the actions' argument;
- the entity types over which each parameter ranges, and the subsumption relation among the types; and
- the entailment relation from obligations to the rights granted in exchange from them.

This is not formalized:

- the interpretation of each action.

This formalization is sufficient to support automated licensing calculations to support designers, developers, integrators, and acquisition analysts.
The task of creating the formalization is not small, but the resulting automation saves substantial work with every licensing analysis.

Testable Leaves

We hypothesize that software oracles can be constructed for most actions in license obligations. Examples are

- an oracle to check for specific warranty disclaimers in source code and
- an oracle to check whether a specific source file is available at a specific URL.

And so forth. In most cases, it will be impractical or impossible to automatically check for *any* condition that would fulfill the obligation (say, for any copyright notice that would serve for LGPLv2.1), but it will be practical to check for a *specific* condition known to fulfill it (say, for one specific copyright notice that serves).

How Subsumption Fits Into License Analysis

In our previous work, we explored license relationships among rights and obligations using Hohfeld jural relations, and described how we automated licensing analyses for specific systems and incorporated the analyses into a software architecture development environment (Alspaugh et al., 2009a, 2011; Alspaugh et al., 2010). The work presented here provides a new and more extensible foundation for those analyses.

For example, LGPLv2.1 (Free Software Foundation, 1999) states,

> You may modify your copy … of the Library … and … distribute such modifications … provided that you … cause the files modified to carry prominent notices stating that you changed the files and the date of any change. (§ II.2¶1)

A specified subsumption relation among actions might classify "Richard Roe distributed `glibc` v1.2.3 source on 2012 July 26" as an instance of the rights action "Distribute WBOL source" (§ II.1¶1) and "Jane Doe placed LGPLv2.1 change notices in `glibc` v1.2.3 on 2012 July 25" as an instance of the obligations action "Insert notices of changes"(§ II.1¶1), which is itself subsumed by rights action "Create WBOL source"(§ II.1¶1; see Figure 1). It still remains to relate the fulfilled obligation of placing the notices to the desired right of distributing the modified source (as well as the other obligations imposed by LGPLv2.1), and presumably to decide whether Jane Doe and Richard Roe were acting jointly so that her fulfilled obligation supported his exercised right.

Legal Interpretations

We believe the interpretation provided by the formalization provides sufficient scope for legal interpretations, based on informal conversations with lawyers and researchers in law, but the choices provided by the formalization do not appear to be a natural expression of the choices a lawyer would make. This will require future work.

Questions We Need Not Ask

For the goals we have set for this research, it is not necessary to be able to answer certain research questions. Examples we have identified are as follows:

- *Given two software licenses, how are they related?*
  We believe the answer for any pair of existing licenses is that they are not equivalent and neither one subsumes the other. Our research indicates that useful comparisons are only possible for individual rights or obligations, or at most, for groups of a few.
- *Can our approach account for delegation?*
  We are not aware of any licenses with provisions that explicitly address delegation of obligations. Licensing is of course a delegation of one or more rights. Our approach does not analyze this kind of delegation beyond a surface level.
- *Would it be advantageous to apply a temporal, deontic, or other specific logic to licensing analysis?*
  Perhaps; such logics might enable the asking of different kinds of questions that we cannot address at present. Our combination of a logic based on Hohfeld (1913) and description logic suffices for our stated goals. We continue to look for contexts in which additional kinds of reasoning would be beneficial.

Future Work

We hypothesize that license-based reasoning about software systems offers benefits in domains beyond that of intellectual property licenses such as LGPL. Our ongoing research program is examining the use of license-like structures for security, envisioning "security licenses" to fulfill the goals of security policies and similar measures but in a more manageable and scalable manner, with direct application to software engineering processes such as open-architecture OSS development of systems integrated from components from many sources. Data licensing is another promising area, especially since data licenses already exist.

Conclusion

We present initial results from an analysis of LGPLv2.1 (Free Software Foundation, 1999) in its entirety, based on earlier work that analyzed high-value areas of a collection eventually numbering 46 licenses. The analysis covers the license textually in several senses:

1. as a grounded-theory analysis chunking and open-coding the entire text;
2. as a higher-level synthesis by which the license text was partitioned a second time (into definitions, rights, obligations, modifiers, and no-effect); and
3. as all LGPLv2.1 actions and the relations among them from which arises the structure of rights and obligations for the license.

The analysis also identified *actions* as the central concept around which license structure is organized. When actions are taken as the fundamental construct, the characteristics of rights and obligations become emergent phenomena arising from the relationships among a license's desired, required, and forbidden actions. The focus on

actions also led us to a more flexible and generalized approach for parameterizing actions and deriving a subsumption relation among them. We extended the subsumption relation to include the actions for the relevant exclusive copyright rights (Figure 1), and to relate the actions for rights and obligations. Grounding the relation in the actions of the exclusive rights proved helpful in distinguishing actual rights and obligations from provisions in the textual guise of rights or obligations but serving the function of modifiers of definitions, rights, and obligations. While no analysis or interpretation of a license can be considered final, the three kinds of coverage achieved and cross-correlated (of the text at both an open-coding and an axial coding level, and of the license's actions supported by a grounding in the copyright exclusive actions) give confidence in the results.

References

[1] Abadi, M., & Cardelli, L. (1996). *A theory of objects*. New York, NY: Springer-Verlag.

[2] Alspaugh, T. A. (2011). Annotated Copy of the GNU Lesser General Public License, version 2.1. Retrieved from http://www.thomasalspaugh.org/pub/osl-sps/lgpl2.1.html

[3] Alspaugh, T. A, Asuncion, H. U., & Scacchi, W. (2009a). Intellectual property rights requirements for heterogeneously-licensed systems. In *Proceedings of the 17th IEEE International Requirements Engineering Conference (RE '09)* (pp. 24–33). Los Alamitos, CA: IEEE.

[4] Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2009b). The role of software licenses in open architecture ecosystems. In *Proceedings of the First International Workshop on Software Ecosystems* (pp. 4–18). Springer, Berlin.

[5] Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2011). Presenting software license conflicts through argumentation. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)* (pp. 509–514). Knowledge Systems Institute, Skokie, IL.

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

[7] Berne Convention, *Berne Convention for the Protection of Literary and Artistic Works.* (1979). Retrieved from the World Intellectual Property Organization website: http://www.wipo.int/treaties/en/ip/berne/

[8] Black Duck Software. (n.d.). Top 20 most commonly used licenses in open source projects. Retrieved from http://www.blackducksoftware.com/oss/licenses

[9] Corbin, J. M., & Strauss, A. C. (2007). *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Thousand Oaks, CA: SAGE.

[10] Free Software Foundation. (1999). GNU Lesser General Public License, version 2.1. Retrieved from http://www.gnu.org/licenses/lgpl-2.1.html

[11] German, D. M., & Hassan, A. E. (2009). License integration patterns: Addressing license mismatches in component-based development. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '09)* (pp. 188–198). New York, NY: ACM.

[12] Gobeille, R. (2008). The FOSSology project. In *Proceedings of the International Working Conference on Mining Software Repositories (MSR '08)* (pp. 47–50). New York, NY: ACM.

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

[14] Otto, P. N., & Antón, A. I. (2007). Addressing legal requirements in requirements engineering. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE '07)* (pp. 5–14). Los Alamitos, CA: IEEE.

[15] Sergot, M. J., Sadri, F., Kowalski, R. A., Kriwaczek, F., Hammond, P., & Cory, H. T. (1986). The British Nationality Act as a logic program. *Communications of the ACM*, *29*(5), 370–386.

[16] U.S. Copyright Act, 17 U.S.C. (1976). Retrieved from http://www.copyright.gov/title17/

Acknowledgements

This research is supported by grant #N00244-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #0808783 from the U.S. National Science Foundation. No review, approval, or endorsement implied.
The authors thank the anonymous reviewers whose insightful suggestions helped us improve the paper.

Licensing Security

Thomas A. Alspaugh & Walt Scacchi

Abstract

There exist legal structures defining the exclusive rights of authors, and means for licensing portions of them to others in exchange for appropriate obligations. We propose an analogous approach for security, in which portions of exclusive security rights owned by system stakeholders may be licensed as needed to others, in exchange for appropriate security obligations. Copyright defines exclusive rights to reproduce, distribute, and produce derivative works, among others. We envision exclusive security rights that might include the right to access a system, the right to run specific programs, and the right to update specific programs or data, among others. Such an approach uses the existing legal structures of licenses and contracts to manage security, as copyright licenses are used to manage copyrights. At present there is no law of "security right" as there is a law of copyright, but with the increasing prevalence and prominence of security attacks and abuses, of which Stuxnet and Flame are among the best known recent examples, such legislation is not implausible. We discuss kinds of security rights and obligations that might produce fruitful results, and how a license structure and approach might prove more effective than security policies.

Introduction

Security mechanisms for implementing software security requirements and policies are often employed on an ad hoc basis rather than in a scalable, organized, and effective manner. Convenient, interactive approaches supported by automated evaluation and guidance are not available because there is no formal basis connecting security requirements and policies with the security mechanisms that are to fulfill them. What is available is a palette of disjoint mechanisms for implementing individual system security features (Loscocco et al., 1998; Spencer et al., 1999) augmented by generalized practices and process standards, such as the following:

1. mandatory access control lists;
2. firewalls;
3. multi-level security;
4. authentication (certificate authorities, passwords);
5. cryptographic support (e.g., public key certificates);
6. encapsulation (including virtualization and hidden rather than public APIs), hardware confinement (memory, storage, port, and external device isolation; Sun, Wang, Zhang, & Stavrou, 2012), and type enforcement capabilities;
7. data content or control signal flow logging/auditing;
8. honey-pots and traps;
9. functionally equivalent but diverse multi-variant software executables (Franz, 2010; Salamat, Jackson, Wagner, Wimmer, & Franz, 2011);

10. security technical information guides (STIGs) for configuring the security parameters for applications (Defense Information Systems Agency, 2011) and operating systems (Sun et al., 2012);
11. secure programming practices (secure coding standards, data type and value range checkin; Seacord, 2008); and
12. standards for development organization processes and practices rather than system security policies (International Standards Organization/International Electrotechnical Commission, 2005).

The reader will note that these mechanisms are *software implementation choices* or *software process choices* rather than *system architectural choices* or *security requirements/policy choices*. Between these mechanisms and a workable concept of a comprehensive security policy for a system or its substantial components, is a gap, with no obvious way to bridge it. Consequently, we note the following practical limitations:

- There is no common framework or conceptual basis with which to integrate and evaluate mechanisms in combination. It is unclear how the various security mechanisms are related and how one may contribute to or interfere with another.
- Guidance is scant for analysts, architects, and developers who need to decide which security mechanism to use where, when, how, and why; and also for integrators and administrators who need to decide how to update the selection of mechanisms and their configuration within a system as security needs and policies evolve.

No satisfactory framework exists in which security requirements and policies can be assembled in hierarchical patterns that can be designed and combined in a system architecture to meet specific high-level security policies and requirements.

We believe there is an opportunity to address security requirements challenges throughout a system architecture using *security licenses*.

In our previous work (Alspaugh, Asuncion, & Scacchi, 2009, 2011; Alspaugh & Scacchi, 2009; Alspaugh, Scacchi, & Asuncion, 2010), we showed how software licenses for the components of a system can be used to guide architectural choices and evaluate rights and obligations for the system as a whole, even when components are governed by different licenses. Using our approach, a system architect can work both down from the top, propagating desired license rights for the system down to individual components to see what license obligations are required to obtain those rights, and up from the bottom, combining license rights and obligations for components and then subsystems into the total rights and obligations for the system. In either direction, our approach identifies any conflicts and mismatches among licenses in the architecture.

We propose the same approach for security licenses. System architects and analysts can select desired security rights, assign an expected security license to each subsystem or

component, and evaluate interactions between these choices at every level from an individual component up to the entire system. Of course, assigning a security license to a component does not guarantee that the component's developer will make it satisfy its security obligations, any more than accepting a component under the GNU General Public License (GPL) guarantees that the system's stakeholders will satisfy the GPL intellectual property (IP) obligations. But assigning a license (whether security or IP) to each component records the assumptions being made about that component and its use, and evaluating those licenses in the context of the system's architecture identifies mismatches and conflicts among those assumptions for that architecture's design choices. When the evaluation is automated, as it is in our work (Alspaugh et al., 2011), it forms the foundation for design guidance with respect to the issues raised by the licenses, and a means for combining the potentially dissimilar licenses to evaluate their overall interaction and effect, and thus the overall interaction and effect of the security mechanisms that are expected to satisfy the obligations and of the security requirements and policies that the rights express.

Security Licenses

In general terms, a security license is analogous to an ordinary software license such as GPL (GNU General Public License; Free Software Foundation, 2007). Software licenses consist of intellectual property (IP) rights granted by the licensor, in exchange for corresponding license obligations imposed on the licensee. A license presents the rights that are offered, and for each right enumerates the obligations that are required in order for that right to be granted. Many of the actions required for the obligations are related to the actions allowed by the rights. This is particularly so for open-source licenses, for which fulfilling some of the obligations requires parts of the rights that are granted. Also, particularly for open-source licenses, the obligations and rights are framed to take effect in an architectural context, with most obligations taking effect with respect to either the component for which rights are granted or component(s) determined by the connectors and architectural topology around that component. Because software licenses are expressed in natural language, the rights and obligations are often presented in an intermingled organization, and much of a license may be devoted to defining terms, classes of entities referred to, and conditions under which the various provisions take effect. But the conceptual structure remains that of a list of rights offered, each in exchange for specific obligations.

Our innovation is to similarly specify components' security rights and obligations, which we can then model, analyze, and support throughout the system's development and evolution, and use to guide its design and instantiation.

There is no "Securityright Act" analogous to the U.S. Copyright Act (1976), or Berne Convention (1979), to define the exclusive security rights of system stakeholders. We present these possible security rights and obligations as an indication of what sorts of actions might be regulated by security licenses for data organized into security compartments and code organized into components.

Some Possible Security Rights

1. The right to read data in compartment T
2. The right to add data to compartment T
3. The right to remove data from compartment T
4. The right to replace component C with another component D
5. The right to update component C to newer version C′
6. The right to revert component C to older version C′
7. The right to add component C in a specified architectural configuration
8. The right to update component C in a specified architectural configuration
9. The right to alter the architectural topology of subcomponent B
10. The right to alter the architecture of system S
11. The right to add security mechanism M in a specified configuration
12. The right to update security mechanism M in a specified configuration
13. The right to remove security mechanism M from a specified configuration
14. The right to delegate security right R
15. The right to read the security license of component C
16. The right to replace the security license L of component C with another security license L′
17. The right to update security license L

Some Possible Security Obligations

1. The obligation for user U to verify his/her identity, by password or other specified authentication process
2. The obligation for user U to have been vetted by authority A to exercise security right R
3. The obligation for user U to be delegated a one-time right by authority A to exercise security right R
4. The obligation for component C to have been vetted by authority A to exercise security right R
5. The obligation for component C to have been vetted by authority A to be the object of security right R
6. The obligation for each component connected to component C to allow it to exercise security right R
7. The obligation for security license L to meet specified criteria
8. The obligation for security license L to be approved by authority A

Effectiveness, Manageability, Evolvability

Consider the case of the development of an open-architecture (OA) system integrating proprietary and open-source components from a variety of producers, most of whom do not coordinate their activities and none of whom are controlled by the organization producing the OA system. From the point of view of ensuring security, this is arguably the worst possible case, but it is an increasingly prevalent development model (Alspaugh et al., 2010). The OA approach gives access to a wide selection of complex components of high quality, and allows the system to evolve as quickly as its integrators can find appropriate new versions or new components and evolve their architecture and shim code to accommodate them.

Since the producers do not coordinate, they are unlikely to use the same security approaches, and indeed may not even publish what those approaches are. To control security in the resulting system, each component is enclosed in a *containment vessel* (Scacchi & Alspaugh, 2012) that isolates the component with a hypervisor ("Xen Hypervisor," n.d.) and mediates all communication with the component (method/function calls, or data streams,) through shim code that monitors and restricts it.

A typical current-day technique (Luo & Du, 2011) for managing security measures is to use *capability lists* to control each component's access to resources such as function calls and data compartments. Each access is delayed briefly while the monitor checks the access against the accessing component's capability list, then blocked if the component was not granted the capability to access that resource. In our experience, each capability list is a text file listing allowed and/or forbidden capabilities, managed manually; new capabilities are typically added to the end of the file. As there appears to be no formal model supporting relationships among capabilities, interactions between capabilities are also identified and managed manually. The text files are detailed, which is a positive aspect, but therefore also long and mind-numbingly tedious, so errors inevitably creep in and are not noticed. Because a capability list has no hierarchy or recursive structure, managing them is not scalable.

A more sophisticated approach is possible using a declarative policy language such as Ponder (Damianou, Dulay, Lupu, & Sloman, 2001) or an ontology-based language such as KAoS (Uszok et al., 2004) that groups capabilities hierarchically, in ontologies (KAoS) or grouped by roles (Ponder). However, they have no provision for organizing capabilities by software components, combined hierarchically into system architectures, and no obvious connection to law.

We contrast the use of security licenses. In some ways, the approaches are similar, in that our candidate security rights are reminiscent of capabilities, and security licenses can also be used to identify and block disallowed operations automatically. However, because many of the actions required for the security obligations are related by subsumption to those granted by the security rights, and many of the obligations are in the context of the component for which corresponding rights are being granted, it is

possible to automatically calculate the interaction of rights and obligations throughout the immediate neighborhood of each component, the subsystem containing the component, and so on, recursively on up to the system as a whole (Alspaugh et al., 2009). Structuring the security policies as licenses gives a form that is more readily accessible to human readers, and helps convey intention and rationale by relating each obligation to the right it contributes toward. Where the security licenses assigned to the components in the architecture conflict or misalign, automated support can identify the provisions in conflict, locate the conflict to the modules involved, and provide explanations showing the architectural chain of effects that led up to the conflict (Alspaugh et al., 2011). Perhaps most importantly, it supports automation of the analysis of interactions between security measures and of the assessment of the system's overall degree and kind of security as a function of the measures taken for each component, group of components, subsystem, and so forth, recursively up to the system as a whole.

Recent Events

Coordinated international attacks on vulnerable software-intensive systems of high value and controlling complex systems are becoming ever more apparent. As the Stuxnet case demonstrates, security threats to software systems are multi-valent, multi-modal, and distributed across independently developed software system components (Falliere, Murchu, & Chien, 2011). Similarly, it is now clear that even physically isolated systems are vulnerable to external security attacks, via portable storage devices like USB drives, modified end-user devices like keyboards and mice (Henning, 2011), and social engineering techniques (Sawers, 2011). New security measures and policy types are required to defend such systems through new threat detection and parrying methods, as well as appropriate active defense mechanisms. What makes a system or system architecture secure changes over time, as new threats emerge and as systems evolve to meet new functional requirements. Consequently, there is need for an approach that can continuously assure the security of complex, evolving systems in ways that are practical and scalable, yet robust, tractable, and adaptable.

The Stuxnet attacks entered through software system interfaces at either the component, application subsystem, or base operating system level, and their goal was to go outside or beneath their entry context. However, all of the Stuxnet attacks on the targeted software system could be blocked or prevented through security capabilities associated with the open software interfaces that would (a) limit access or evolutionary update rights lacking proper authorization, as well as (b) "sandboxing" (i.e., isolating) and holding up any evolutionary updates (the attacks) prior to their installation and run-time deployment. Furthermore, as the Stuxnet attack involved the use of corrupted certificates of trust from approved authorities as false credentials that allowed evolutionary system updates to go forward, it seems clear that additional preventions are needed that are external to, and prior to, their installation and run-time deployment. The development-, installation-, and configuration-time rights and obligations previously addressed extend the ordinary run-time benefits of security

licenses to defend against development-, distribution-, configuration-, and update-time attacks.

Exclusive Security Rights

If there could be legally defined and protected exclusive security rights, what would they be? We nominate the following candidates for discussion:

1. the right of the owner of a copy of a system to replace, update, or revert any of its components;
2. the right of the owner of a copy of a system to add or remove components or otherwise alter its architectural topology;
3. the right of the owner of a copy of a system to replace or update the security license of the system or any of its components; and
4. the right of the owner of a copy of a system to alter its user Input/Output streams or ephemeral data. (We envision that persistent data may fall into a different category of protected entity.)

As with the exclusive copyright rights, the owner of a right may license all or part of it to someone else in exchange for obligations, for example, to allow a trusted system provider to automatically install certain kinds of updates.

References

[1] Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2009). Intellectual property rights requirements for heterogeneously-licensed systems. In *Proceedings of the17th IEEE International Requirements Engineering Conference (RE '09)* (pp. 24–33). Los Alamitos, CA: IEEE.

[2] Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2011). Presenting software license conflicts through argumentation. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)* (pp. 509–514). Knowledge Systems Institute, Skokie, IL.

[3] Alspaugh, T. A., & Scacchi, W. (2009). Heterogeneously-licensed system requirements, acquisition, and governance. In *Proceedings of the Second International Workshop on Requirements Engineering and Law (RELAW '09)* (pp. 13–14). Washington, DC: IEEE Computer Society.

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

[5] Berne Convention for the Protection of Literary and Artistic Works. (1979). Retrieved from http://www.wipo.int/treaties/en/ip/berne/

[6] Damianou, N., Dulay, N., Lupu, E., & Sloman, M. (2001). The Ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks* (pp. 18–39). Retrieved from http://pdf.aminer.org/000/545/721/the_ponder_policy_specification_language.pdf

[7] Defense Information Systems Agency. (2011). *Android 2.2 (Dell) security technical implementation guide (STIG)*. Retrieved from Defense Information Systems Agency, http://iase.disa.mil/stigs/net_perimeter/wireless/smartphone.html

[8] Falliere, N., Murchu, L. O., & Chien, E. (2011). *W32.Stuxnet dossier* (Technical report). Retrieved from http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[9] Franz, M. (2010). E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms (NSPW '10)* (pp. 7–16). Retrieved from http://www.ics.uci.edu/~franz/Zurich/MassiveScaleDiversity.pdf

[10] Free Software Foundation. (2007). *GNU General Public License, version 3*. Retrieved from http://www.gnu.org/licenses/gpl-3.0.html

[11] Henning, E. (2011). Attack of the computer mouse. *The H Security*. Retrieved from http://www.h-online.com/security/news/item/Attack-of-the-computer-mouse-1270018.html

[12]International Standards Organization/International Electrotechnical Commision (ISO/IEC, 2005). International standard 27001, retrienved from http://www.27001-online.com/

[13] Loscocco, P. A., Smalley, S. D., Muckelbauer, P. A., Taylor, R. C., Turner, S. J., & Farrell, J. F. (1998). The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference (NISSC '98)* (pp. 303–314). National Technical Information Service, Silver Springs, MD.

[14] Luo, T., & Du, W. (2011). Contego: Capability-based access control for web browsers. In *Proceedings of the Fourth International Conference on Trust and Trustworthy Computing (TRUST '11)* (pp. 231–238). Heidelberg, Germany: Springer-Verlag Berlin.

[15] Salamat, B., Jackson, T., Wagner, G., Wimmer, C., & Franz, M. (2011). Runtime defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing*, *8*(4), 588–601.

[16] Sawers, P. (2011). US Govt. plant USB sticks in security study, 60% take the bait. *The Next Web (TNW)*. Retrieved from http://thenextweb.com/insider/2011/06/28/us-govt-plant-usb-sticks-in-security-study-60-of-subjects-take-the-bait

[17] Scacchi, W., & Alspaugh, T. A. (2012, July). Advances in the acquisition of secure systems based on open architectures. *Journal of Cybersecurity & Information Systems,* January 2013, (to appear).

[18] Seacord, R. C. (2008). *CERT C secure coding standard*. New York, NY: Addison-Wesley.

[19] Smalley, S. (2012). *The case for security enhanced (se) android*. Presentation at the 2012 Android Builder's Summit, Redwood Shores, CA.

[20] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., & Lepreau, J. (1999). The Flask Security Architecture: System support for diverse security policies. In *Proceedings of the Eighth USENIX Security Symposium (SSYM '99)* (pp. 123–139). Retrieved from http://www.cs.utah.edu/flux/papers/flask-usenixsec99.pdf

[21] Sun, K., Wang, J., Zhang, F., & Stavrou, A. (2012). SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity OSes. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)*. Retrieved from http://cs.gmu.edu/~astavrou/research/Secure_Switching_ndss2012.pdf

[22] U.S. Copyright Act, 17 U.S.C. (1976). Retrieved from http://www.copyright.gov/title17/

[23] Uszok, A., Bradshaw, J. M., Johnson, M., Jeffers, R., Tate, A., Dalton, J., & Aitken, S. (2004). KAoS policy management for semantic web services. *IEEE Intelligent Systems*, *19*(4), 32–41.

[24] Xen hypervisor. (no date). Retrieved from http://xen.org/products/xenhyp.html

Acknowledgements

This research is supported by grant #N00244-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #0808783 from the U.S. National Science Foundation. No review, approval, or endorsement implied.

THIS PAGE INTENTIONALLY LEFT BLANK

2003 - 2012 Sponsored Research Topics

Acquisition Management

- Acquiring Combat Capability via Public-Private Partnerships (PPPs)
- BCA: Contractor vs. Organic Growth
- Defense Industry Consolidation
- EU-US Defense Industrial Relationships
- Knowledge Value Added (KVA) + Real Options (RO) Applied to Shipyard Planning Processes
- Managing the Services Supply Chain
- MOSA Contracting Implications
- Portfolio Optimization via KVA + RO
- Private Military Sector
- Software Requirements for OA
- Spiral Development
- Strategy for Defense Acquisition Research
- The Software, Hardware Asset Reuse Enterprise (SHARE) repository

Contract Management

- Commodity Sourcing Strategies
- Contracting Government Procurement Functions
- Contractors in 21st-century Combat Zone
- Joint Contingency Contracting
- Model for Optimizing Contingency Contracting, Planning and Execution
- Navy Contract Writing Guide
- Past Performance in Source Selection
- Strategic Contingency Contracting
- Transforming DoD Contract Closeout
- USAF Energy Savings Performance Contracts
- USAF IT Commodity Council
- USMC Contingency Contracting

Financial Management

- Acquisitions via Leasing: MPS case

- Budget Scoring
- Budgeting for Capabilities-based Planning
- Capital Budgeting for the DoD
- Energy Saving Contracts/DoD Mobile Assets
- Financing DoD Budget via PPPs
- Lessons from Private Sector Capital Budgeting for DoD Acquisition Budgeting Reform
- PPPs and Government Financing
- ROI of Information Warfare Systems
- Special Termination Liability in MDAPs
- Strategic Sourcing
- Transaction Cost Economics (TCE) to Improve Cost Estimates

Human Resources

- Indefinite Reenlistment
- Individual Augmentation
- Learning Management Systems
- Moral Conduct Waivers and First-term Attrition
- Retention
- The Navy's Selective Reenlistment Bonus (SRB) Management System
- Tuition Assistance

Logistics Management

- Analysis of LAV Depot Maintenance
- Army LOG MOD
- ASDS Product Support Analysis
- Cold-chain Logistics
- Contractors Supporting Military Operations
- Diffusion/Variability on Vendor Performance Evaluation
- Evolutionary Acquisition
- Lean Six Sigma to Reduce Costs and Improve Readiness
- Naval Aviation Maintenance and Process Improvement (2)
- Optimizing CIWS Lifecycle Support (LCS)
- Outsourcing the Pearl Harbor MK-48 Intermediate Maintenance Activity

- Pallet Management System
- PBL (4)
- Privatization-NOSL/NAWCI
- RFID (6)
- Risk Analysis for Performance-based Logistics
- R-TOC AEGIS Microwave Power Tubes
- Sense-and-Respond Logistics Network
- Strategic Sourcing

Program Management

- Building Collaborative Capacity
- Business Process Reengineering (BPR) for LCS Mission Module Acquisition
- Collaborative IT Tools Leveraging Competence
- Contractor vs. Organic Support
- Knowledge, Responsibilities and Decision Rights in MDAPs
- KVA Applied to AEGIS and SSDS
- Managing the Service Supply Chain
- Measuring Uncertainty in Earned Value
- Organizational Modeling and Simulation
- Public-Private Partnership
- Terminating Your Own Program
- Utilizing Collaborative and Three-dimensional Imaging Technology

A complete listing and electronic copies of published research are available on our website: www.acquisitionresearch.net

THIS PAGE INTENTIONALLY LEFT BLANK

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL
555 DYER ROAD, INGERSOLL HALL
MONTEREY, CALIFORNIA 93943

www.acquisitionresearch.net