UCI-AM-16-010

ACQUISITION RESEARCH PROGRAM SPONSORED REPORT SERIES

Achieving Better Buying Power through Acquisition of Open Architecture Software Systems Volume II

Understanding Open Architecture Software Systems: Licensing and Security Research and Recommendations

6 January 2016

Dr. Walt Scacchi Dr.

Thomas A. Alspaugh

Institute for Software Research University

University of California, Irvine

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.

Abstract

This research focuses on continuing investigation and refinement of techniques for identifying and reducing the costs, streamlining the process, and improving the readiness of future workforce for the acquisition of complex software systems. Emphasis was directed at identifying, tracking, and analyzing software component costs and cost reduction opportunities within the acquisition life cycle of open architecture (OA) systems for Web-based and mobile devices, where such systems combine best-of-breed software components and software products lines (SPLs) that are subject to different IP license and cybersecurity requirements. The investigation focuses on four project work activities:

- Investigating the interactions between software system acquisition guidelines and processes, and the cost consequences of alternative software system architectures incorporating different mixes of OSS and CSS components subject to different licenses within secure OA SPLs [ScA08, ScA12b, ScA13a, ScA13b, ScA13c]. This entails exploring the balance between development, verification, and validation of software licenses and security rights, as well as the software component/license costs while managing the development and evolution of OA systems at design-time, build-time, and release and run-time.

- Developing formal foundations for establishing acquisition guidelines program managers can use in reduced cost acquisition of software-intensive systems that rely on development and deployment of secure OA systems using OSS and SPL technology and processes [AIS10, AIS13, ScA11, ScA12a, ScA12b, ScA13a, ScA13b, ScA13c].

- Continuing to develop concepts contributing to the emerging design of an automated approach supporting acquisition of secure OA systems by (a) determining their conformance to acquisition guidelines/policies, contracts, and related license management issues, and (b) giving future acquisition workforce support and insights to properly review, approve, and manage the acquisition of complex systems that incorporate cost-sensitive acquisition of OA systems and software components [AIS10, ScA11, ScA12a, ScA12b, ScA13a, ScA13b, ScA13c].

- Documenting the investigation, foundations, and results of the research in: (a) a technical Final Report delivered to the Technical Point of Contact at NPS; (b) a research presentation at the 11th Annual Acquisition Research Conference, in Monterey, CA, May 2014; (c) a progress report with the OSD sponsor and others of interest within the OUSD (AT&L) offices; and (d) related research venues and publications, including periodic research progress reports.

THIS PAGE LEFT INTENTIONALLY BLANK

About the Authors

Dr. 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 2011, he served as co-chair for the 33rd International Conference on Software Engineering—Practice Track, and in 2012, he served as general co-chair of the 8th IFIP International Conference on Open Source Systems.

Dr. Walt Scacchi, Senior Research Scientist
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
E-mail: wscacchi@ics.uci.edu

Dr. Thomas 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.

Dr. Thomas Alspaugh, Project Scientist
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
E-mail: thomas.alspaugh@acm.org

THIS PAGE LEFT INTENTIONALLY BLANK

UCI-AM-16-010

ACQUISITION RESEARCH PROGRAM
SPONSORED REPORT SERIES

Achieving Better Buying Power through Acquisition of Open Architecture Software Systems Volume II

Understanding Open Architecture Software Systems: Licensing and Security Research and Recommendations

6 January 2016

Dr. Walt Scacchi Dr.

Thomas A. Alspaugh

Institute for Software Research University

University of California, Irvine

THIS PAGE LEFT INTENTIONALLY BLANK

Executive Summary

The goal of this research was to create a new approach to address Better Buying Power challenges in the acquisition of software systems for the Department of Defense. Program managers, acquisition officers, and contract managers will increasingly be called on to review and approve choices between functionally similar low or no cost open source software components, and commercially priced closed source software components, to be used in the design, implementation, deployment, and evolution of open architecture (OA) systems. We seek to make this a simpler, more transparent, and more tractable process. Such a process must identify, track, and analyze software component costs throughout the system life cycle, and be easy to reuse for different system application domains, in order to realize cost reductions and improve acquisition workforce capabilities. Our recent research demonstrates how complex OA systems can be designed, built, and deployed with alternative components and connectors resulting in functionally similar system versions, to satisfy overall system security requirements and individual system component intellectual property (IP) requirements [DODOSA13, SEI13]. Our next step, described in this two volume Final Report, is to identify, track, and analyze software component costs associated with different types of component IP licenses when acquiring OA systems, and to do so in ways that highlight opportunities for cost reduction. We believe our results will be applicable to enterprise software systems in other government agencies and industrial firms, as well as to enterprise and mission-critical systems for the DoD community.

This research focuses on continuing investigation and refinement of techniques for identifying and reducing the costs, streamlining the process, and improving the readiness of future workforce for the acquisition of complex software systems. Emphasis was directed at identifying, tracking, and analyzing software component costs and cost reduction opportunities within acquisition life cycle of open architecture (OA) systems for Web-based and mobile devices, where such systems combine best-of-breed software components and software products lines (SPLs) that are subject to different IP license and cybersecurity requirements.

The Department of Defense, other government agencies, and most large-scale business enterprises continually seek new ways to improve the functional capabilities of their software-intensive systems. The acquisition of OA systems that can adapt and evolve through replacement of functionally similar software components is an innovation that can lead to lower cost systems with more powerful functional capabilities. This research seeks to identify and analyze how software component costs for Web-based and mobile devices, component IP license and cybersecurity requirements interact to drive down (or drive up) total system costs across the system acquisition life cycle. The availability of such new scientific knowledge and technological practices can give rise to more effective expenditures of public funds and improve the effectiveness of future software-intensive systems used in government and industry. Thus, the principal purpose of this research supports and advances a public purpose.

Finally, our principal research results are documented in two volumes.

Volume I includes four contributions. In Chapter 1 we summarize details of our research efforts in the past 12 months. These efforts have been well received in presentations to different audiences, including within the larger Defense community, and the Federal Government more broadly. In particular, our research results have been picked up for use within the *Assembled Capabilities Working Group* (ACWG, previously identified as the DoD Widget Working Group, through early 2014), under the guidance of the C3CB (Command, Control, Communications, and Business Systems) office within the OUSD (AT&L). This effort was facilitated through collaboration with many people from The MITRE Corporation, who along with the C3CB office are working in support of the Defense Intelligence Information Enterprise (DI2E) and related mission partners. Summary presentations that have been publicly shared resulting from our research appear in Chapter 2, 3, and 4. Chapter 2 includes the abstract and slide deck that were presented at the 2014 Acquisition Research Symposium

(May 2014). Chapter 3 is the slide deck from MITRE-ATARC Workshop in Washington, DC (August 2014) addressing Cost-Sensitive Acquisition of Open Architecture Software Systems for Mobile Devices. Chapter 4 is the slide deck from the Federal Mobile Computing Summit also held in Washington, DC (August 2014). Further, in response to many requests for additional information on our research approach, methods, and results, we have compiled an integrated report of ten chapters that bring together our research results that span from 2007 through this project year's effort. These chapters address: (1) Cost-Sensitive Acquisition of Open Architecture Software Systems; (2) Open Architectures for Software Systems; (3) License Challenges for Open Architectures; (4) Software License Legal Foundations; (5) Automating License Analysis; (6) Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems; (7) Processes in Securing Open Architecture Software Systems; (8) Addressing Challenges in the Acquisition of Secure Software Systems with Open Architectures; (9) Ongoing Software Development without Classical Requirements; (10) Discussion and Recommendations. Specific recommendations that follow from our research that address the question, *How best to improve and streamline acquisition processes for secure OA systems,* can be identified as follows (and elaborated in Chapter 10, Volume II, this Report):

- *Encourage the adoption of acquisition business models in open source formats*
- *Encourage the development, (re)use and refinement of open source models of acquisition processes*
- *Develop and employ techniques for streamlining acquisition of secure OA systems, via*
    - *Acquisition process measurement and assessment*
    - *Acquisition process redesign and evolution*
    - *Design new acquisition processes*
    - *Cost management as an acquisition process design element*

These technical details, research integration, and more are found within Volume II of this Final Report. Last, it is our opinion that the compilation and integration of concepts, techniques, and materials presented in Volume II is a work in progress, and so it will benefit from ongoing refinement going forward, hopefully to be shown as part of our new (2015-16) acquisition research project now in progress.

Overall, we welcome any comments or questions on our research efforts, results, or recommendations.

References

[DoDOSA13] Department of Defense Open Systems Architecture Data Rights Team (2013). *DoD Open Systems Architecture Contract Guidebook for Program Managers* (Version 1.1). DoD, June 2013. https://acc.dau.mil/OSAGuidebook

[SEI13] Software Engineering Institute (2013). Managing Intellectual Property in the Acquisition of Software-Intensive System, November.

Table of Contents

Chapter 1.

Cost-Sensitive Acquisition of Open Architecture Software Systems

Chapter 1.

Cost-Sensitive Acquisition of Open Architecture Software Systems

Abstract

This chapter focuses on introducing our ongoing investigation and refinement of techniques for identifying and reducing the costs, streamlining the process, and improving the readiness of future workforce for the acquisition of complex software systems. Emphasis is directed at introducing our approach to identifying, tracking, and analyzing software component costs and cost reduction opportunities within acquisition life cycle of open architecture (OA) systems [DoDOSA11] , where such systems combine best-of-breed software components and software products lines (SPLs) that are subject to different intellectual property (IP) license requirements.

The Department of Defense, other government agencies, and most large-scale business enterprises continually seek new ways to improve the functional capabilities of their software-intensive systems. The acquisition of OA systems that can adapt and evolve through replacement of functionally similar software components is an innovation that can lead to lower cost systems with more powerful functional capabilities. Our research identifies and analyzes how software component costs and IP license requirements interact to drive down (or drive up) total system costs across the system acquisition life cycle. The availability of such new scientific knowledge and technological practices can give rise to more effective expenditures of public funds and improve the effectiveness of future software-intensive systems used in government and industry. Thus, a goal of this book and the chapters that follow is to support and advance a public purpose through acquisition research and results.

Overview

Interest within the U.S. Department of Defense (DoD) and military services in free and open source software (OSS) first appeared in the past five or so years [cf. Bol03]. More recently, it has become clear that the U.S. Air Force, Army, and Navy have all committed to a strategy of acquiring software-intensive systems across the board that require or utilize an “open architecture” (OA) and “open technology” (OT) which may incorporate OSS technology or OSS development processes [HeS07].

Across the three military services within the DoD, OA means different things and is seen as the basis for realizing different kinds of outcomes. Thus, it is unclear whether the acquisition of a software system that is required to incorporate an OA as well as utilize OSS technology and development processes [cf. Whe07] 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 or reconcile such differences in expectations and outcomes from OA strategies in each service or across DoD. Yet there is little explicit guidance or reliance on systematic empirical studies for how best to develop, deploy, and sustain complex software-intensive military systems in the different OA and OSS presentations and documents that have so far been disseminated [cf. Wea07]. Instead, what mostly exists are narratives that serve to provide ample motivation and belief into the promise and potential of OA and OSS without consideration of what socio-technical challenges may lie ahead in realizing OT, OA, and OSS strategies.

Acquisition officers are familiar with the challenges of acquiring systems that meet the necessary requirements with regard to correct behavior: the correctness of the overall system depends on the correctness of its components and how they are interconnected; correctness is a relative quality, in that a system may meet its behavioral requirements to a greater or lesser degree, but almost by definition a system is never completely correct, and its degree of correctness cannot be definitely established in a finite time; a lack of correctness has an effect when that part of the system is executing; and the correctness of a system in meeting its requirements is determined, by engineers and the system's users, through testing it and using it. Openness is both similar to and different from correctness, however. We argue that the openness of a system depends, like correctness, on the system's components, how they are interconnected, and how they are configured into an overall software system architecture. Unlike for correctness, however, a system may be completely open, or may fail to be open in various ways; and because the software elements that define a system are finite and enumerable, its openness can in principle be determined. Also unlike correctness, a system is either open or not open even when it is not operating, and DoD may pay the consequences of a lack of openness (in the form of license fees) before the system is ever used, or even if it is never used. Finally, unlike for correctness, openness may ultimately by the province of lawyers and policy makers, not of engineers or users.

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(s) apply to the configured system as a whole. This challenge comprises not only evaluating an existing system, but planning 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 kinds of licenses, each of which may affect a system in different ways, and because there are a number of different kinds of OSS components and ways of combining them that affect the licensing issue, a first necessary step is to understand kinds of software elements that constitute a software architecture, and what kinds of licenses may encumber these elements or their overall configuration. OA seem to simply suggest software system architectures incorporating OSS components and open application program interfaces (APIs). But not all software system architectures incorporating OSS components and open APIs will produce OA, since OA depend on: (a) how/why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, (c) whether the copyright (Intellectual Property--IP) licenses assigned to different OSS components encumber all/part of a software system's architecture into which they are integrated, and (d) many alternative architectural configurations and APIs that may or may not produce an OA [cf. AIA07, Sca07]. Subsequently, we believe this can lead to situations in which if program acquisition stipulates a software-intensive system with an OA and OSS, then the resulting software system may or may not embody an OA. This can occur when the architectural design of a system constrains system requirements—that is, what requirements can be satisfied by a given system architecture, when requirements stipulate specific types or instances of OSS (e.g., Web browsers, content management servers) to be employed, or what architecture style [BCK03] is implied by given system requirements. Thus, given the goal of realizing an OA and open technology strategy [cf. HeS07] together with the use of OSS components and open APIs, it is unclear how to best align program acquisition, system requirements, software architectures, and OSS license regimes to achieve this goal.

It now appears there are a new set of requirements that are emerging that will need to be addressed in any acquisition of a software-intensive system that is stipulated to employ an OA that accommodates OSS components or connectors. Identifying specific requirements for a given program acquisition or system development contract can benefit from consideration of the the following guidelines for how best to realize an OA:

- Determining how much openness is required or desired.
- Identifying guidelines and incentives for software development contractors that encourage them to develop, provide, and distribute/deploy OA systems with OSS components, connectors, and configuration that minimize conflicting OSS IP license obligations.
- Determining the restrictions, if any, that the OSS IP licenses used by different software system components, connectors, or configurations within a OA system.
- Identifying alternative OSS component, connector, or configuration candidates that may satisfy a specified overall system architecture.
- Determining scenarios that help reveal whether there are OSS IP licensing conflicts for a given set of OSS components, connectors, or configuration.
- Identifying and analyzing any OSS IP licensing obligations that must be satisfied for the resulting system to be available for redistribution.
- Identifying and validating OSS IP license conformance criteria for configured systems intended for redistribution.
- Determining when OA systems are to be constructed from OSS or proprietary software components, each subject to distinct and possibly conflicting IP license obligations, the resulting systems becomes heterogeneously-licensed, thus how best to determine whether, where, and to what extent these software IP license conflicts can be resolved.
- Determining how cybersecurity requirements are mediated or transmuted by software IP licenses, as well as whether these requirements may better be expressed in a similar manner to IP obligations and rights.

The Role of OA in Improving the Effectiveness and Reducing System Cost

The Department of Defense, other government agencies, and most large-scale business enterprises continually seek new ways to improve the functional capabilities of their software-intensive systems with lower acquisition costs. The acquisition of OA systems that can adapt and evolve through replacement of functionally similar software components is an innovation that can lead to lower cost systems with more powerful functional capabilities. OA system acquisition, development and deployment are thus seen as an approach to realizing Better Buying Power (BPP) goals for lowering system costs while improving competition.

Our research identifies and analyzes in the chapters that follow how new software component technologies, like large OSS applications or small apps and widgets for Web-based and/or mobile devices, along with their IP license and cybersecurity requirements interact to drive down (or drive up) total system costs across the system acquisition life cycle. The availability of such new scientific knowledge and technological practices can give rise to more effective expenditures of public funds and improve the effectiveness of future software-intensive systems used in government and industry. Thus, a goal of this presentation is to explore new ways and means for achieving cost-sensitive acquisition of OA software systems, as well as identifying factors that can further decrease or increase the costs of such systems at this time.

We begin by briefly reviewing to identify a set of recent trends in the development of OA software systems that intend to develop more capable OA systems. These trends include the transition to adoption of small-form factor software components as distinct applications ("apps") and widgets that exploit modern Web capabilities. We then turn to examine some key goals of the BBP 2.0 and 3.0 initiatives that direct attention to adoption of OA system development practices that affect acquisition practices. Next, we identify a new set of emerging challenges to achieving BBP through OA software systems. We then identify three new practices to realize the cost-effective acquisition of OA Software systems.

Recent Trends Affecting Better Buying Power through OA Systems

We find there are four broad trends that mediate the cost-effectiveness and buying power of emerging OA system acquisition efforts. These include: (a) the move towards shared, multi-party acquisition and agile development of new OA systems across compatible software ecosystems; (b) exploitation of new software component technologies compatible with Web and mobile devices; (c) growing diversity of cybersecurity challenges to address during system development; (d) new software development business models for app/widget development and deployment. Each is examined in turn.

A. Multi-party acquisition and development system ecosystems – Many in the Defense community seek to embrace the acquisition and development of agile command and control (C2) and related enterprise systems [GBC14, GMH13, GuW12, RBC12, ScA13c, SBN12]. Such systems are envisioned to arise from the assembly and integration of OA system elements (application components, widgets, content servers, networking elements, etc.) within a software ecosystem of multiple producers, integrators, and consumers who may supply or share the results of their efforts. The assembly and integration of system elements produces "C2 system capabilities" (C2SCs). Our purpose is to identify how our approach to the design of secure OA systems can be aligned with this emerging vision for agile C2 system development and adaptive deployment. Along the way we focus on design of OA system capability involving office productivity components that must be configured as a secure C2SC.

The design and development of agile C2 systems follows from two sets of principals: one set addressing guidelines/tenets for multi-party engineering (MPE) of C2 system components; the other set addressing attributes of agile and adaptive ecosystems (AAE) for producing C2SCs or C2 system elements. For brevity, we identify the principals for MPE and AAE, as they are more fully explained elsewhere [RBC12], but we do so in ways that foreshadow and more clearly align with our approach that follows in later sections.

Multi-Party Engineering Tenets:

- Provide small system components that can be rapidly developed, and accommodate different functionally equivalent variants, or functionally similar versions (software product lines).
- Certify components are consistent with "shared agreements" regarding security requirements, system architecture, data semantics, production and integration processes or process constraints, and other aspects of mission-specific or mission-common domain models.
- Supply diverse C2 system components via a market of component producers or system integrators.
- Assemble and integrate C2SCs from components available in the market that are consistent with relevant shared agreements.
- Provide feedback from C2 system users to component producers or capability integrators to improve market efficiency and effectiveness.

Adaptive and Agile Ecosystem Attributes:

- Encourage and sustain a software ecosystem that is agile (supports assembly and integration C2SC) from components in market, and adaptive (supports substitution of functionally similar component versions or functionally equivalent component variants), in line with user feedback.
- Component markets are federated so as to accommodate sharing, reuse, or trading of components across system integrators or user organizations.
- Shared agreements serve as a basis for enabling multi-party collaboration in system development, integration, and evolution/sustainability.
- Production, integration, or post-deployment support for components or C2SCs must be viable for small businesses or large, as well as promoting market diversity and effectiveness.
- Consumer/user organizations seek to manage portfolios of components or C2SCs that collectively improve mission effectiveness, agility and adaptiveness, while reducing costs.

Subsequently, to help understand what we mean by a software ecosystem, we use Figure 1 to represent where different parties are located across a generic software ecosystem, and the supply networks or multi-party relationships that emerge to enable the software producers to develop and release products that are assembled and integrated by system integrators for delivery to end-user organizations.

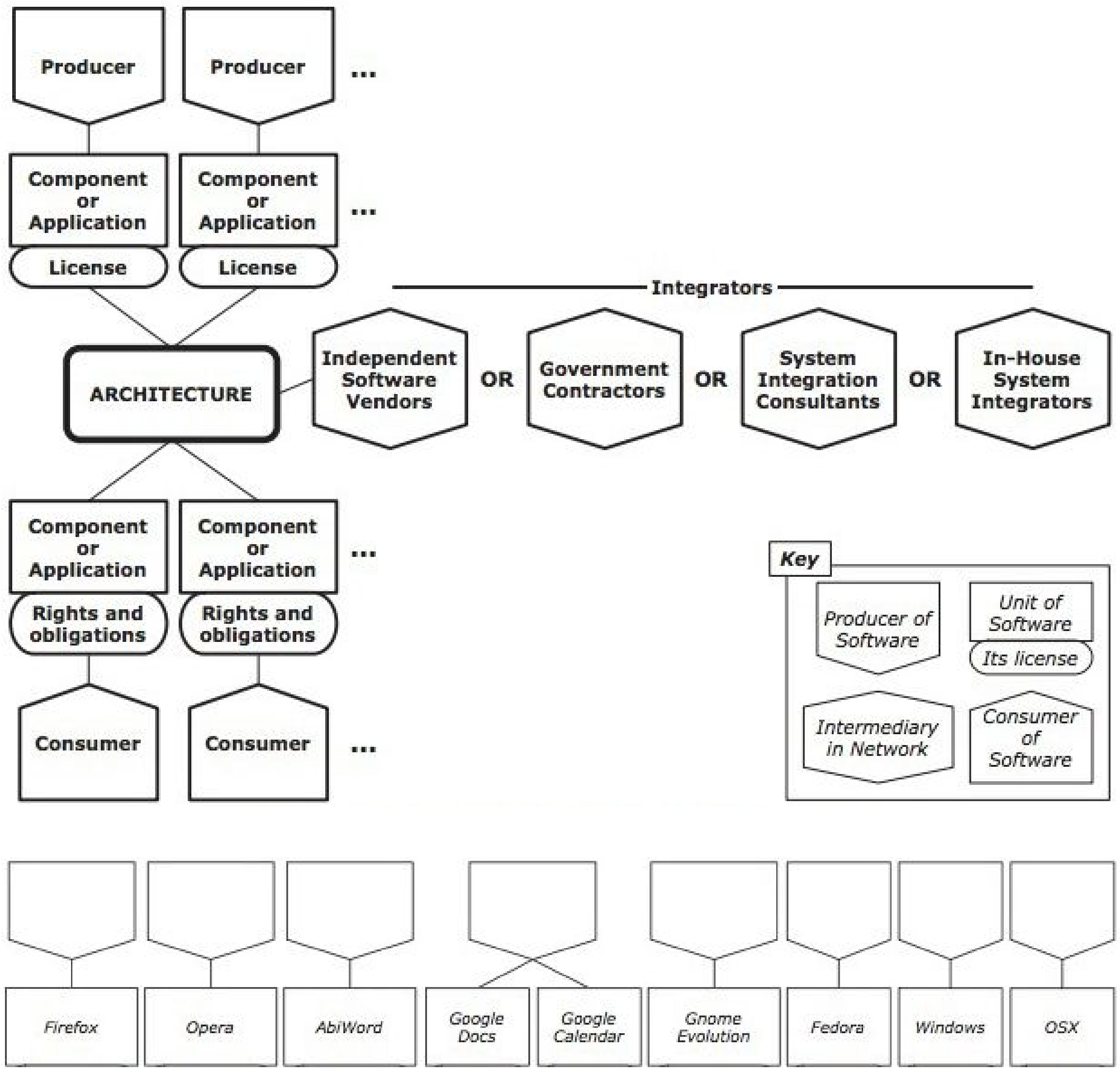


Figure 1. A generic software ecosystem supply network (upper part), along with a sample elaboration of producers, software component applications, and licenses for an OA system components they employ (lower part). Details in Chapter 6..

As noted, OA system components can include software applications (apps) and widgets. Widgets are lightweight, single-purpose web-enabled applications that users can configure to their specific needs [Giz11, GMH13, ScA13b]. Widgets can provide summary information or a limited view into a larger application that can be used alongside related widgets provides an integrated view, as required by users.

The lower part of Figure 1 also identifies where elements of shared agreements like IP licenses or cybersecurity requirements enter into the ecosystem, and how the assembly of components into a configured system or subsystem architecture by system integrators effectively (and perhaps unintentionally) determines which IP license or cybersecurity obligations and rights get propagated to consumer or end-user organizations. Agreement terms and conditions acceptable to consumer/end-user organizations flow back to the integrators. This helps reveal where and how shared agreements will mix, match, mashup, or encounter semantic mis-matches at the system architecture level, which is one reason why we use (and advocate) explicit OA system models.

A substantial number of development organizations are adopting a strategy in which a software-intensive system is developed with an OA whose components may be 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.

Similarly, a move towards MPE and AAE substantiates a path towards *decentralized OA system development, integration, and deployment* [DoD12, Giz11, SBN12]. This decentralization will in turn engender acquisition and development of heterogeneously-licensed systems (HLS), whereby different software components (apps, widgets) will be subject to different IP licenses [AAS12, ASA10], as well as to different cybersecurity requirements [DAG14, ScA12b, ScA13a, ScA13b, ScA13c]. This in turn implies that such components, their IP licenses, and cybersecurity requirements will be subject to ongoing evolution across a diversity of methods, shown in Figure 2 [ScA12a, ScA13b].

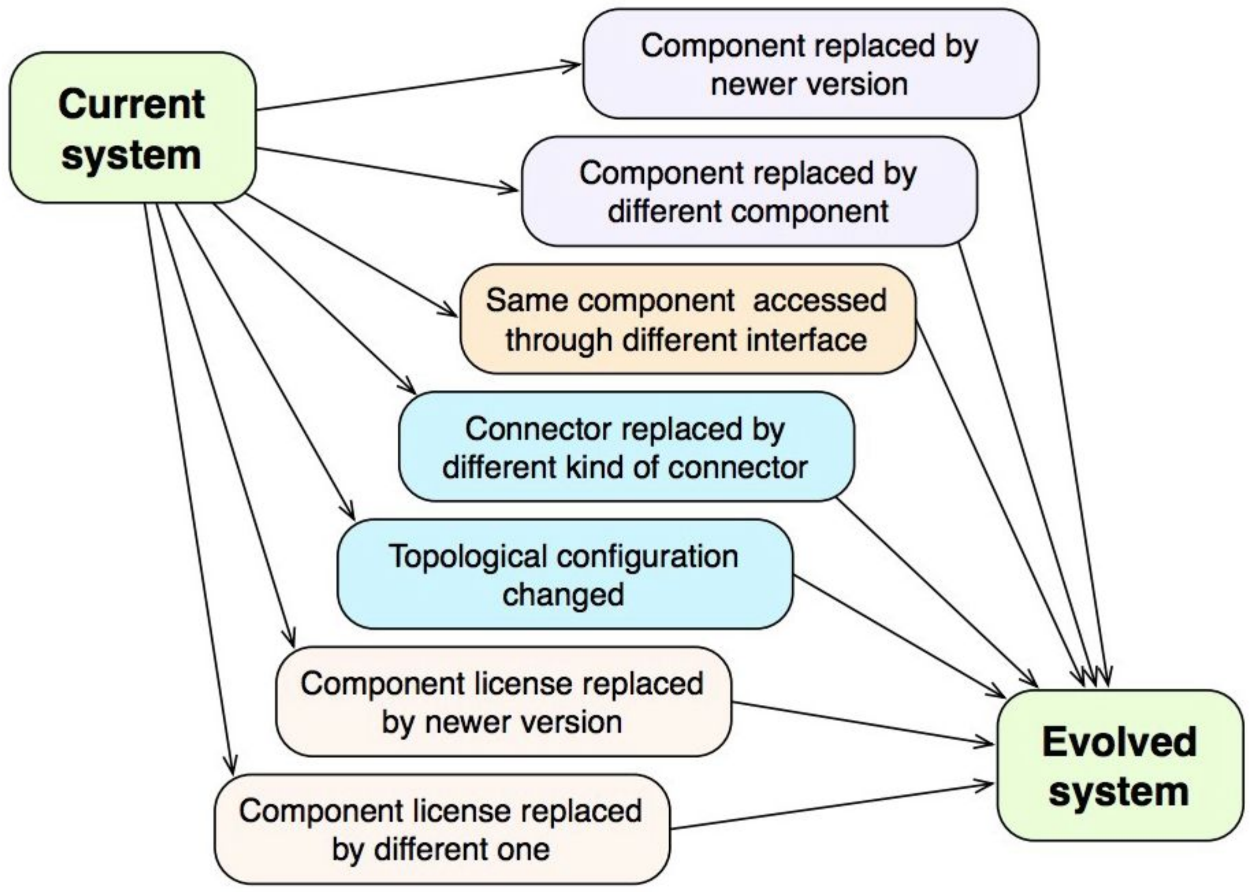

Current system
Component replaced by newer version
Component replaced by different component
Same component accessed through different interface
Connector replaced by different kind of connector
Topological configuration changed
Component license replaced by newer version
Component license replaced by different one
Evolved system

Figure 2. Different kinds of common evolutionary changes that arise during OA software component development, deployment and sustained usage (see Chapter 6)].

Heterogeneity, decentralization, cybersecurity and evolutionary dynamics will come to interact during OA system acquisition, development, and deployment. These in turn will create a new generation of challenges for the acquisition workforce, in terms of training, new work and contract management practices, and need for automated assistance to track and manage oversight of policy compliance (e.g., for alignment with BPP and cybersecurity assessment). Without automated assistance, it appears that the acquisition workforce will be overwhelmed with technical details that interact with acquisition, development, and/or system integration contracts and software component IP licenses and cybersecurity requirements. Otherwise, these conditions suggest that acquisition management practices can complicate acquisition [GBC14], and thus potentially mitigate the benefits of BBP that can arise from MPE and AAE for C2 systems.

B. Moving towards shared development of Apps and Widgets as OA system components – Future OA systems for agile C2 may configured by system integrators, end-user organizations, or warfighters in the field. This would be accomplished through access to online repositories of software apps or user-interface widgets. The Ozone Widget Framework (OWF), a government open source software (GOSS) effort that is central to such agile OA system development. The OZONE family of products, includes the OWF and the OZONE Marketplace, the marketplace being an online repository whose operation is similar in kind to the online app stores by Apple and Google [ScA13b]. These products are built to fit the needs of human centered fusion activities in network centric warfare environments. The OZONE family of products is designed as a presentation layer toolkit that can be rapidly deployed in a variety of mission contexts ranging from strategic planning to enabling the creation of a real-time common operational picture and situation awareness applications. Figure 3 displays examples of OWF-based widgets operating in a Web browser, while Figure 4 shows OWF widgets deployed for use on a mobile device.

C. Growing diversity of challenges in cybersecurity – New types of software components like apps and widgets must be developed, deployed, and sustained in ways compatible with existing cybersecurity requirements. They must also be later adapted to accommodate emerging cybersecurity requirements that are not yet apparent. For example, there is growing interest in accommodating not just mobility, but also “Bring Your Own Device” (BYOD) capabilities. BYOD suggests that end-users and warfighters are bringing their own mobile devices with themselves into the field to support their mission. However, BYOD clearly exacerbates the technical challenges of cybersecurity assurance, often in ways that cannot be readily anticipated, as when independently developed component co-evolve in conflict to one another [Wei14]. Nonetheless, acquisition policy necessitates cybersecurity vulnerability and exposures be addressed [DAG14]. But at present, it is unclear what new kinds of requirements these new OA system components bring to the acquisition workforce. For example, a move to adopt mobile apps and/or mobile widgets means these OA system components must pass though an application security process for “vetting” these components.

Vetting entails establishing what cybersecurity requirements are to be verified, how they are to be validated, as well as where, when and by whom these activities should be performed. One approach is to assume such vetting can be performed by a centralized authority, such as by the operator of the Ozone Marketplace. But it is not clear there will ever only be one such authority. Instead, if we foresee multiple marketplaces, which are already appearing both in GOSS and industrial online settings, then the acquisition workforce will be challenged in how best to determine which cybersecurity requirements must be addressed, validated, and compliance certified, as well as by whom and how often.

A move to widgets also presents new kinds of cybersecurity challenges when two or more widgets are configured together with one or more apps to create a mashup that provides an agile system capability. This situation refers to the technical challenges of inter-widget communication. Such component-component communication can be technically realized in different ways, such as via ad hoc, “open standards,” or publish-subscribe messaging interfaces, as well as whether point-to-point or as configured through a dynamic processing mashup [CFG13, End13]. While OA system guidance from the BBP 2.0 may stipulate reliance on “open standards” style widget interfaces and communications patterns be used, widget communication/interface standards/interfaces are still very new technologies and techniques.

Thus, it is unclear which will survive and be widely adopted [End13a]. Similarly, knowledge about their proper usage is unclear, and thus is not yet ready for compliance assessment within current acquisition practices. The technical challenge is further complicated when apps/widgets are acquired from different online marketplaces. Different marketplaces may rely on different schemes for specification and interchange of shared data semantics between autonomously developed components. This in turn hinges on the expertise of OA system integrators, end-users, or warfighters to recognize how, where, and when the semantics of technical data interchange arise and to what consequences via component-component API alignments (to avoid mis-matches), data type representations, data formats (e.g., "CSV" vs. .xls vs. XML), data naming conventions (for resource discovery vs data modeling ontology), data range value limits, exceptional values, data-flow control signals, etc. These are still new technical problems that are yet to be readily resolved or to have development/usage guides.

D. New business models for OA software component development and use – New business models imply differentiated IP licenses and contracting practices. Given our discussion up to this point, along with reference to our recent acquisition research studies [AAS12, ScA11, ScA12b, ScA13b], this means different obligations and rights will be transferred from component producers to system integrators and end-user organizations. Some licenses are "buy and pay now," while others are "free now, pay later, based on usage," others are "many organizations (e.g., PEOs) will share purchase costs," and so forth.

Acquisitions of new kinds of OA system components allow for new business models. These include new models for software component producers, system integrators, and end-user organizations. For example, new software and OA system development business models for software app/widget development and deployment include (in no particular order): franchising; enterprise licensing; metered usage; advertising supported; subscription; free component, paid service/support fees; federation reciprocity for shared development; collaborative buying; donation; sponsorship; free/open source software (e.g., Government OSS – *GOSS*); and others [Hanf13]. Further, this list is not exhaustive; instead, it is only representative.

In contrast, for end-user organizations that involved in agile development of OA system components, or an integrated system capability, there is a need to developed and codify their own business models regarding OA software component development or system integration. These business models are constituted through "shared agreements" that allow for sharing the cost of component or integrated capability development and cybersecurity assurance vetting across multiple parties (e.g., multiple Program Offices). However, these shared agreements are also a core part of emerging MPE/AAE development practices. These agreements must convey how OA component development or system integration costs and security assurance will be shared, as well as how they will be sustained in the presence of interacting software component development, deployment, and evolution processes and practices [ScA13a]. Shared agreements denote the obligations the participating organizations are willing to accept, in order to realize the provided rights they need. So shared agreements can be expressed and assessed in the same manner, and with the same analysis tools and techniques, as IP licenses and cybersecurity requirements [ScA13b, ScA13c].

Software acquisition costs easily become difficult to predict/manage given diversity of business models, IP licenses, and implied software component cybersecurity assessment. Development/usage cost sharing agreements can further complicate determination of development cost, costs shares across organizations, and system costs over time as business models, component licenses, and cybersecurity assessment requirements evolve [ScA12a, ScA13a].

What kind of expertise do we expect the acquisition workforce to need in order to make adoption of “component-based system capabilities” (including for mobile devices) agile, adaptive, and practical across different commercial/governmental software marketplaces or ecosystems? What kinds of acquisition guidance is needed for articulating and streamlining Shared Agreements between multiple organizations participating in shared OA component development and cybersecurity assurance? What kinds of acquisition management practices and analysis tools are needed for the acquisition workforce to insure cost savings and BBP in such settings? Addressing these questions is beyond the scope of this paper, but these questions require follow-on acquisition research to resolve and answer.

Better Buying Power Goals

Better Buying Power (http://bbp.dau.mil/) is part of DoD's mandate to *do more without more* by implementing best practices in acquisition. BBP identifies seven areas of focus that group a larger set of 36 initiatives that offer the potential to restore affordability in defense procurement and improve defense industry productivity. One of the seven areas focuses on promoting or increasing competition, and this area includes an initiative to “enforce open system architectures and effectively manage technical data rights” [DAU12]. Technical data rights pertain to two categories of Intellectual Property (IP): they refer to the Government's rights to (a) technical data (TD – e.g., product design data, computer databases, computer software documentation); and (b) computer software (CS – e.g., source code, executable code, design details, processes, and related materials). These rights are realized through IP licenses provided by system product or service providers (e.g., software producers) to the Government customer, so long as the customer fulfills the obligations stipulated in the license agreement (e.g., to indicate how many software users are authorized to use the licensed product or service according to a fee paid).

As already noted, our acquisition research has focused on issues addressing OA systems and IP licenses since 2008 [ScA08], as well as forward to the acquisition of secure OA systems for command and control (C2) and enterprise information systems [ScA11, ScA12b, ScA13b], where security requirements can be expressed in a manner similar to IP obligations and rights. Therefore, here we turn to identify how a sample of different goals of BBP 2.0 initiatives interact or relate to the trends and challenges examined so far in this paper. The BBP goals are highlighted, then followed by a brief examination.

- *Increase competition* – One central purpose for acquiring OA systems is to increase the likelihood of competition among system producers who can provide software components that can be replaced by similar offerings by other component producers. We demonstrate how this can work when system architectures are explicitly modeled, and their software components and interconnections are similarly specified in an open manner [AAS12, ScA12a].
- *Adopt OA systems that utilize standardized interfaces* – Open system architectures that can accommodate common components from alternative producers requires that the components utilize standardized interfaces, whether in the form of open Application Program Interfaces (APIs), standard data exchange protocols, and standard data representations, formats, and meta-data [ScA08]. But also noted earlier, app and widget components at present have a plethora of standardized interfaces, and it is unclear which will survive, be sustained, be widely adopted (inside/outside of DoD), and be evolved [End13a].
- *Utilize open source software components where appropriate to reduce costs* – another aspect of openness that OA systems embrace and DoD policy accepts is to utilize system components developed as open source software (OSS) [DIS12]. Utilization of OSS components, along with composing OA systems that incorporate OSS and closed, proprietary components, does require careful attention to the management and analysis of multiple IP licenses that apply to different OA system components, as well as determining what overall IP and/or cybersecurity rights and obligations cover the overall system [AAS12, ScA12a], especially for C2 systems [AAS12, ScA13b, ScA13c].
- *Increase small business roles and opportunities* – one way to increase competition in the realm of OA systems is to identify where smaller scale software applications (apps) or widgets can be utilized, which might be produced by small businesses or startup ventures which dominate much of the online markets for Web-based or mobile device apps/widgets. Small businesses may further be advantaged by their utilization of OSS infrastructure components, platforms, or remote services, since large commercial contractors may not see sufficient profit margins to develop proprietary alternatives. So OA systems that accommodate OSS components that can integrate custom apps/widgets into innovative system capabilities (C2SC), may then realize new opportunities for DoD customers. Other small business opportunities may similarly arise for such ventures that focus on emerging cybersecurity assessment or tool development services.
- *Use technical development phase for true risk reduction and rapid prototyping* – In looking forward, there is potential interest in seeing the BPP initiative evolve to also address risk as an implicit cost driver. This might allow or innovative ways and means to reduce emerging risks through accelerated or "look ahead" system acquisition and development approaches that emphasize increased reliance on rapid prototyping. This kind of rapid prototyping might even be performed by appropriately trained end-users or warfighters. A move towards OA systems for Web-based and mobile devices that rely on apps/widgets retrieved from online marketplaces, that can be composed through interpretive software program "scripting" and mashup techniques, is a clear example of

this [End13, GMH13 GuW12, ScA13a]. Thus, it is not surprising to find such emerging techniques being investigated and assessed for possible production of new C2 capabilities [GBC14, GMH13, ScA13b].

- *Do more without more* – an overall summary of the BBP effort is focusing attention of how to make acquisition more agile, to do more without more, and to develop a new generation acquisition workforce that can enact acquisition processes that are thin and flexible when needed, yet robust and cost-effective, while also being amenable to continuous improvement. This is indeed a real challenge to fulfill, and beyond the scope of what current acquisition practices are likely to achieve without targeted investment in acquisition improvement research. To be clear, one just needs to consider emerging opportunities (and potential asymmetric cybersecurity threats) that arise through the desire to develop next-generation C2SC that are to be composed from apps/widgets that can operate on Web-based/mobile devices. What are the best processes or practices for acquiring, developing, and sustaining deployed systems that are to be built using these new software technologies (e.g., apps/widgets for mobile devices)? How should these processes and practices be adapted to accommodate personal devices (e.g, Apple iPhones, Android tablet, Microsoft Mobile Phone, Blackberry 10 phone) that individual warfighters, joint force troops, or contracted service providers bring with them into the battlespace? How must acquisition processes be best adapted to accommodate and rely on software supply chains that arise around consumer-oriented app marketplaces as possible ways/means for *doing more* (e.g., rapidly prototyping warfighter composable C2 app/widget mashups [GMH13]) *without more* (e.g., warfighters who bring their own mobile computing devices for use in C2 contexts) [GBC14]? Once again, these are critical questions to address and resolve through new acquisition research and supporting technology development.

Conclusions

This chapter focused on introducing our ongoing investigation and refinement of techniques for identifying and reducing the costs, streamlining the process, and improving the readiness of future workforce for the acquisition of complex software systems. Emphasis was directed at introducing our approach to identifying, tracking, and analyzing software component costs and cost reduction opportunities within acquisition life cycle of open architecture (OA) systems, where such systems combine best-of-breed software components and software products lines (SPLs) that are subject to different intellectual property (IP) license requirements. The Department of Defense, other government agencies, and most large-scale business enterprises continually seek new ways to improve the functional capabilities of their software-intensive systems. The acquisition of OA systems that can adapt and evolve through replacement of functionally similar software components is an innovation that can lead to lower cost systems with more powerful functional capabilities. Our research described through the following chapters identify and analyze how software component costs and IP license requirements interact to drive down (or drive up) total system costs across the system acquisition life cycle. The availability of such new scientific knowledge and technological practices can give rise to more effective expenditures of public funds and improve the effectiveness of future software-intensive systems used in government and industry.

References

[AIA07] Alspaugh, T.A and Antón, A.I., (2007). Scenario Support for Effective Requirements, *Information and Software Technolog*y, 50(3), 198-220.

[AAS12] Alspaugh, T.A, Asuncion, H. and Scacchi, W. (2012). The Challenge of Heterogeneously Licensed Systems in Open Architecture Software Ecosystems, S. Jansen, S. Brinkkemper, and M. Cusumano (Eds.), *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry,* Edward Elgar Publishing, 103-120, Northampton, MA.

[ASA10] Alspaugh, T.A, Scacchi, W., and Asuncion, H. (2010). Software Licenses in Context: The Challenge of Heterogeneously Licensed Systems, *Journal of the Association for Information Systems*, 11(11), 730-755, November 2010.

[BCK03] Bass, L., Clements, P., and Kazman, R., (2003). *Software Architecture in Practice*, 2nd Edition, Addison-Wesley Professional, New York.

[Bol03] Bollinger, T., (2003). *Use of Free and Open-Source Software (FOSS) in the U.S. Department of Defense*, The MITRE Corporation, 2 January. Available at http://www.terrybollinger.com/dodfoss/dodfoss_html/index.html

[CFG13] Chudnovsky, O., Fischer, C. Gaedke, M. and Pietschmann (2013). Inter-Widget Communication by Demonstration in User Interface Mashups. *Web Engineering*, Springer-Verlag, Lecture Notes in Computer Science, Vol. 7977, 502-505.

[DAG14] *Defense Acquisition Guidebook* (2014). CVE--Common Vulnerabilities and Exposures. Chapter 13.7.3.1.4, https://acc.dau.mil/CommunityBrowser.aspx?id=492079#13.7.3.1.4 accessed April 2014.

[DAU12], Defense Acquisition University (2012). *Open Systems Architecture and Technical Data Rights...Management Approaches*, http://bbp.dau.mil/docs/Open%20Systems%20Architecture%20and%20Technical%20Data%20Rights%20.%20.%20.%20Management%20Approaches.pdf accessed 30 October 2012.

[DIS12], Defense Information Systems Agency (2012). *DOD Open Source and Community Source Software Development in Forge.mil*, SoftwareForge Document ID – doc26066doc26066 http://www.disa.mil/News/Conferences-and-Events/DISA-Mission-Partner-Conference-2012/~/media/Files/DISA/News/Conference/2012/DoD_Open_Source_Community_Forge.pdf accessed 30 October 2012.

[DoD12] Department of Defense (2012). *Joint Operational Access Concept*, Version 1.0, 17 January 2012, http://www.defense.gov/pubs/pdfs/JOAC_Jan%202012_Signed.pdf

[DoDOSA11], Department of Defense Open Systems Architecture (2011). *Contract Guidebook for Program Managers*, Vol. 0.1, December, https://acc.dau.mil/OSAGuidebook

[End13] Endres-Niggemeyer, B. (2013). The Mashup Ecosystem, in *Semantic Mashups: Intelligence Reuse of Web Resources*, Springer, 1-50.

[End13a] Endres-Niggemeyer, B. (2013). Mashups Live on Standards, in *Semantic Mashups: Intelligence Reuse of Web Resources*, Springer, 51-89.

[GBC14] George, A., Bowers, A., Galdorisi, G., Hszieh, S., Morris, M., and Raney, C. (2014). DoD Application Store: Enabling C2 Agility, *Proc. 19Th Intern. Command and Control Research and Technology Symposium*, Paper-104, Alexandria, VA, June 2014.

[GMH13] George, A., Morris, M., Galdorisi, G., Raney, C., Bowers, A., and Yetman, C. (2013) Mission Composable C3 in DIL Information Environments using Widgets and App Stores. *Proc. 18Th Intern. Command and Control Research and Technology Symposium*, Paper-036, Alexandria, VA, June 2013.

[GuW12] Guertin, N. and Womble, B. (2012). Competition and the DoD Marketplace, *Proc. 9th Acquisition Research Symposium*. Vol. 1, 76-82, Naval Postgraduate School, Monterey, CA.

[Giz11] Gizzi, N. (2011). Command and Control Rapid Prototyping Continuum (C2RPC) Transition: Bridging the Valley of Death, *Proceedings 8th Annual Acquisition Research Symposium,* Vol. 1, Naval Postgraduate School, Monterey.

[Han13] Hanf, D. (2013). *MPE/AAE Business Model Framework Overview*. Mitre Corporation, personal communication, July 2013.

[HeS07] Herz, J.C. And Scott, J., (2007). COTR Warriors: Open Technologies and the Business of War, *The DoD Software Tech News*, 10(2), 3-6, June.

[RBC12] Reed, H., Benito, P., Collens, J., and Stein, F. (2012). Supporting Agile C2 with an Agile and Adaptive IT Ecosystem, *Proc. 17th Intern. Command and Control Research and Technology Symposium* (ICCRTS), Paper-044, Fairfax, VA, June 2012.

[Sca07] Scacchi, W., (2007). Free/Open Source Software Development: Recent Research Results and Methods, in M. Zelkowitz (Ed.), *Advances in Computers*, 69, 243-295.

[ScA08] Scacchi, W. and Alspaugh, T., (2008). Emerging Issues in the Acquisition of Open Source Software within the U.S. Department of Defense, *Proc. 5th Acquisition Research Symposium*, NPS-AM-08-036, Naval Postgraduate School, Monterey, CA, May.

[ScA11] Scacchi, W. and Alspaugh, T., (2011). Advances in the Acquisition of Secure Systems Based on Open Architectures, *Proc. 8th Acquisition Research Symposium*, Vol. 1, Naval Postgraduate School, Monterey, CA.

[ScA12a] Scacchi, W. and Alspaugh, T., (2012a) Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems, *Journal of Systems and Software*, 85(7), 1479-1494, July 2012.

[ScA12b] Scacchi, W. and Alspaugh, T., (2012b). Addressing Challenges in the Acquisition of Secure Software Systems with Open Architectures, *Proc. 9th Acquisition Research Symposium*, Vol. 1, 165-184, Naval Postgraduate School, Monterey, CA.

[ScA13a] Scacchi, W. and Alspaugh, T. (2013a). Processes in Securing Open Architecture Software Systems, *Proc. 2013 Intern. Conf. Software and System Processes*, San Francisco, CA, May 2013.

[ScA13b] Scacchi, W. and Alspaugh, T.A. (2013b). Streamlining the Process of Acquiring Secure Open Architecture Software Systems, *Proc. 10th Annual Acquisition Research Symposium*, Monterey, CA, 608-623, May 2013.

[ScA13c] Scacchi, W. and Alspaugh, T.A. (2013c). Challenges in the Development and Evolution of Secure Open Architecture Command and Control Systems, *Proc. 18Th Intern. Command and Control Research and Technology Symposium*, Paper-098, Alexandria, VA, June 2013.

[SBN12] Scacchi, W., Brown, C. and Nies, K. (2012). Exploring the Potential of Virtual Worlds for Decentralized Command and Control, *17th. Intern. Command and Control Research and Technology Symposium (ICCRTS), Paper-096, Fairfax, VA, June 2012*

[Wea07] Weathersby, J.M., (2007). Open Source Software and the Long Road to Sustainability within the U.S. DoD IT System, *The DoD Software Tech News*, 10(2), 20-23, June.

[Wei14] Weir, M. (2014). BYOD Topic: How Complicated Can Calendars Be? *J. Cybersecurity and Information Systems*, 2(1). 18-19.

[Whe07] Wheeler, D.A., (2007). Open Source Software (OSS) in U.S. Government Acquisitions, *The DoD Software Tech News,* 10(2), 7-13, June.

Chapter 2.

Open Architectures for Software Systems

Chapter 2.

Open Architectures for Software Systems

Abstract

This chapter explores and describes open archtiecture techniques for specifying and modeling OA software systems. These systems are composed and configures using different types of software elements, including components (modules) with explicit interfaces, and connectors that configure systems, sub-systems, or system of systems, through component-interface interconnections. The chapter also describes how OA software systems can be formed into software product lines utilizing different selections of specified component types. Such an approach thus serves to provide a strong foundation for aligning OA concepts that are central to Better Buying Power acquisition initiatives, with other advantages that enable software component and OA reuse, across diverse platforms and application domains.

Introduction: Understanding open software architecture concepts

Open architecture (OA) software development is a customization technique that we observed introduced by Oreizy [Ore00] to enable 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. [ScA08]). Using this approach can lower development costs and increase reliability and function. 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*.

A system intended to embody an open architecture using open source software (OSS) technologies and APIs does not clearly indicate what possible mix of software elements may be configured into such a system. To help explain this, we first identify what kinds of software elements are included in common software architectures whether they are open or closed [cf. BCK03].

Software source code components – these include the computer programs that direct the intended computation, calculation, control flow, and data manipulation. These are programs for which the source code is open for access, review, modification, and possible redistribution by their developers. However, there are at least four different forms of computer programs these days.

- *standalone programs* – these are the computer programs that we have long understood, often as isolated systems or monolithic applications that accept data inputs, manipulate and transform this data, and produce outputs (calculated results, information displays, emit control signals to devices, etc.) under user or system administered control.
- *libraries, frameworks, or middleware* – these are collections of software functions no one of which is typically a standalone program. Such software is often expected to be routinely reused in many different systems or applications. This software may also be used to provide a layer of abstraction that hides source code implementation details so as to improve subsequent software portability, or to hide alternative software implementations.
- *inter-application script code* – this software is used to combine independent programs together by associating their respective inputs, outputs, and control variables. This software is sometimes called, "glue code" to suggest its primary use is to connect programs together through the use of "pipes" and/or "filters" which control or modulate the directed flow of information between the associated programs. Such scripts may be as short a a single line of code, but on the other hand, they can be as large as thousands (even hundreds of thousands) source lines of code.
- *intra-application script code* – this software is similar in spirit to inter-application script code, except the focus is on organizing, controlling, and manipulating input and output data/presentations from remote Web services/repositories for view and end-user interaction at the human-computer interface. Popular Web application systems like the Firefox Web browser may be scripted to provide animated user interfaces coded in languages like Javascript, ActionScript, or PhP to create Rich Internet Applications [Fel07] or "mashups" [NeC06, CFG13, End13, End13a], all of which may be available in online app stores [ScA13, GGM14]. Such scripts may be as short as a single line of code, but on the other hand, they can be as large as thousands (even tens of thousands) source lines of code. However, custom intra-application software languages may also be designed to create domain-specific languages (e.g., XUL for Firefox Web browser [Fel07]) for rapid construction of persistent or disposable software functions (or macros), which enable increased software development productivity or end-user programming.

Executable components -- These are programs for which the software is in binary form, 5 and its source code may not be open for access, review, modification, and possible redistribution. Executable binaries are rarely treated as open since they may also be viewed as "derived works" [Ros05] that result from the compilation or interpretation of software source code which may not be available, or may be proprietary. Executable components are widespread and common in every computing system, even in OSS systems. However, executable components may also only become part of a system during its execution through dynamic (or run-time) linking. Finally, though their binary form makes them available for execution through external linkage to some other program, such form also makes figuring out what they do very difficult, if they have little/no documentation available.

- *Application program interfaces/APIs* – These software interfaces are generally not programs that can be executed, but they enable software system developers to access their functionality without direct access to their source code. The availability of externally visible and accessible APIs to which independently developed components can be connected to is the minimum required to form an “open system” [Meyers and Obendorf 2001]. Oftentimes the APIs are treated as if they enable direct access to the otherwise hidden software, but a closed software system may employ a layer of abstract APIs as “shims” that better align multiple program interfaces or security barriers that seek to protect disclosure of private or proprietary information. Such information may include the details of actual software function interfaces (which may be designated as “trade secrets”), or hidden software functions that may only be known to software developers with secure, restricted code access.
- *Software connectors* – 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. These may include software technologies that constitute a “software bus” for plugging in independent software modules (programs or functions), network protocols that enable and control the flow of data between remote programs across a LAN or Internet, or even a database management system (DBMS) that is used to enable data sharing and storage among programs connected to the DBMS. The High Level Architecture (HLA) is an example of a software connector scheme [KWD00], as are CORBA, Microsoft's .NET, and Enterprise Java Beans.
- *Configured system or sub-system* – These are software systems built to conform to an explicit architectural specification. They include software source code/binary components, APIs, and connectors that are organized in a way that may conform to a known “architectural style” such as the Representational State Transfer [FiT02] for Web-based client-server applications, or may represent an original or ad hoc architectural pattern [BCK03]. All of the software elements, and how they are arranged and interlinked, can all be specified, analyzed, and documented using an Architecture Description Language [BCK03] and ADL-based support tools. Beyond this, any or all of the software elements in a configured system or sub-system may be OSS or not. In contrast to a derived work, a configured system or sub-system is considered as a “collective work” and as such is subject to its own copyright and license protection as intellectual property, whether open or closed [Ros05, StL04]. However, such intellectual property declaration cannot employ a license regime on the overall system that supercedes or controverts the license protections/obligations of the individual software elements that 6 constitute the configured system or sub-system.

Figure 1 (originally from [ScA08]) provides an overall view of an archetypal software architecture for a configured system that includes and identifies each of the software elements above, as well as including open source (e.g., Gnome Evolution) and closed source software (WordPerfect) components.

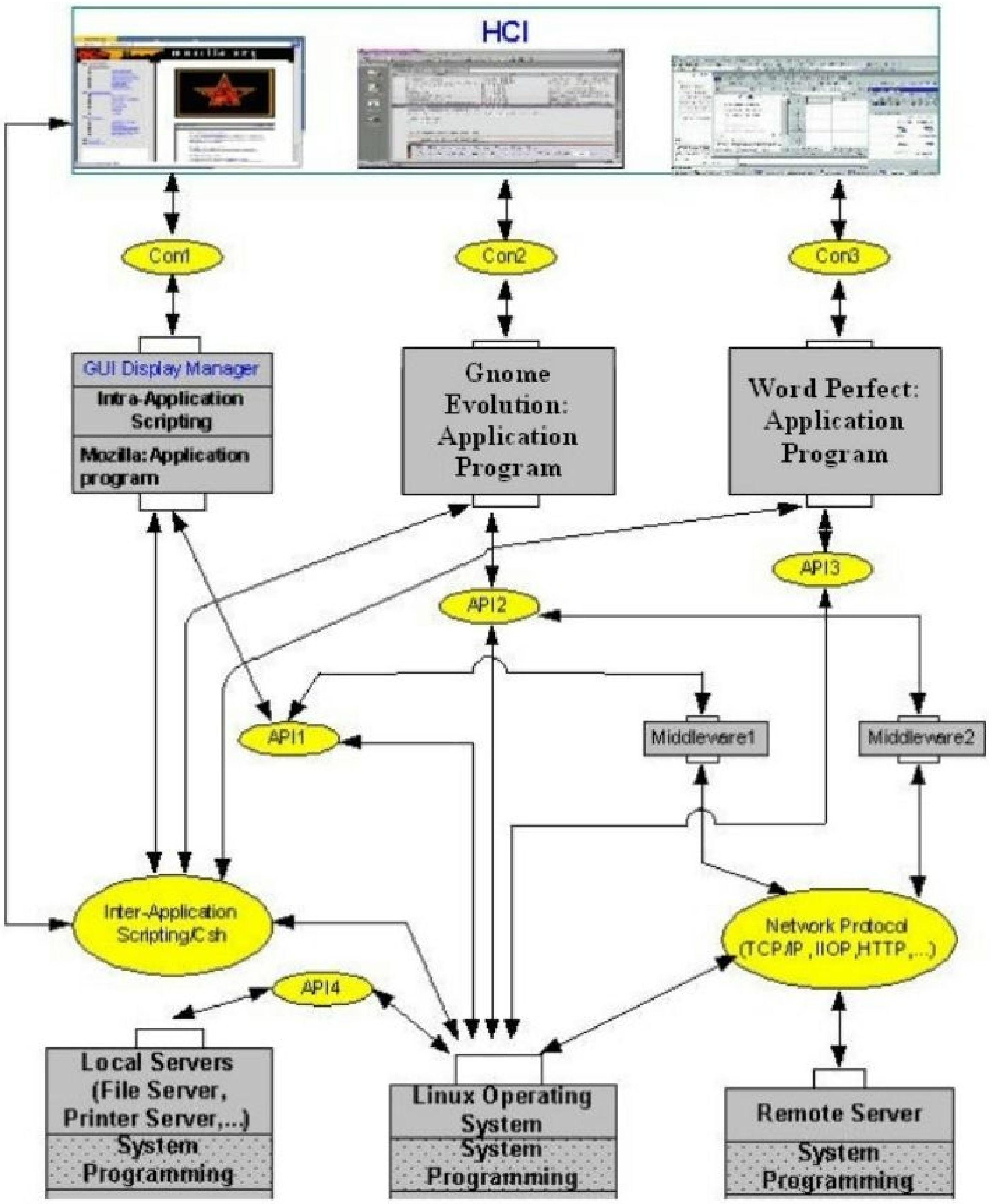
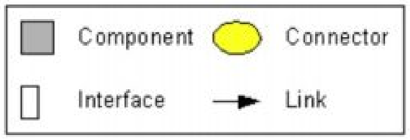
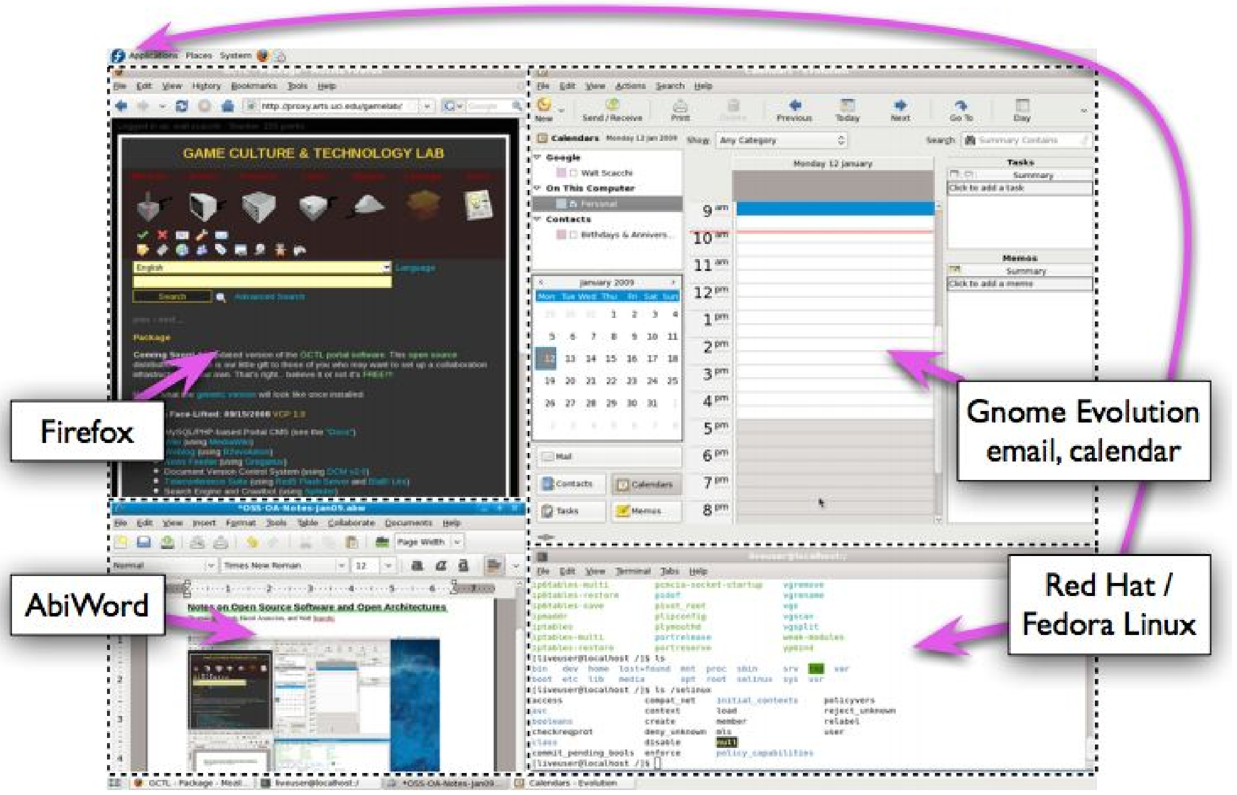

HCI
Con1
Con2
Con3
GUI Display Manager
Intra-Application Scripting
Mozilla: Application program
Gnome Evolution: Application Program
Word Perfect: Application Program
API1
API2
API3
API4
Middleware1
Middleware2
Inter-Application Scripting/Csh
Network Protocol (TCP/IP, IIOP, HTTP,...)
Local Servers (File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

Figure 1. Software components, connectors, interfaces arranged in an overall software system configuration. Components, connectors, and overall system configuration may be subject to different software licenses.



In simple terms, the configured system In Figure 1 consists of software components (grey boxes in the Figure) that include a Mozilla Web browser, Gnome Evolution email client, and WordPerfect word processor that run on a Linux operating system that can access file, print, and other remote networked servers (e.g., Apache Web server).

The components in Figure 1 are interrelated through a set of software connectors (ellipses in the Figure) that connect the interfaces of software components (small white boxes attached to a component) that are linked 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, this simple architecture raises a number of OSS licensing issues that mitigate the extent of openness that is realized in a configured OA.


Firefox
GAME CULTURE & TECHNOLOGY LAB
Gnome Evolution email, calendar
AbiWord
Notes on Open Source Software and Open Architectures
Red Hat / Fedora Linux
Calendars - Evolution

```
ip6tables-multi         pcmcia-socket-startup   vgremove
ip6tables-restore       pcdef                   vgrename
ip6tables-save          pivot_root              vgs
ipmaddr                 plipconfig              vgscan
iptables                plymouthd               vgsplit
iptables-multi          portrelease             weak-modules
iptables-restore        portreserve             ypbind
[liveuser@localhost /]$ ls
bin   dev  home  lost+found  mnt  proc  sbin     srv  tmp  var
boot  etc  lib   media       opt  root  selinux  sys  usr
[liveuser@localhost /]$ ls /selinux
access                  compat_net    initial_contexts    policyvers
avc                     context       load                reject_unknown
booleans                create        member              relabel
checkreqprot            deny_unknown  mls                 user
class                   disable       null
commit_pending_bools    enforce       policy_capabilities
[liveuser@localhost /]$
```

Figure 2. A rendering of an OA software system run-time implementation

Figure 3 shows a high-level view of a reference architecture that includes all the kinds of software elements listed above. This reference architecture has been instantiated in a number of configured systems that combine OSS and closed source components. 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 [ISR06], 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.

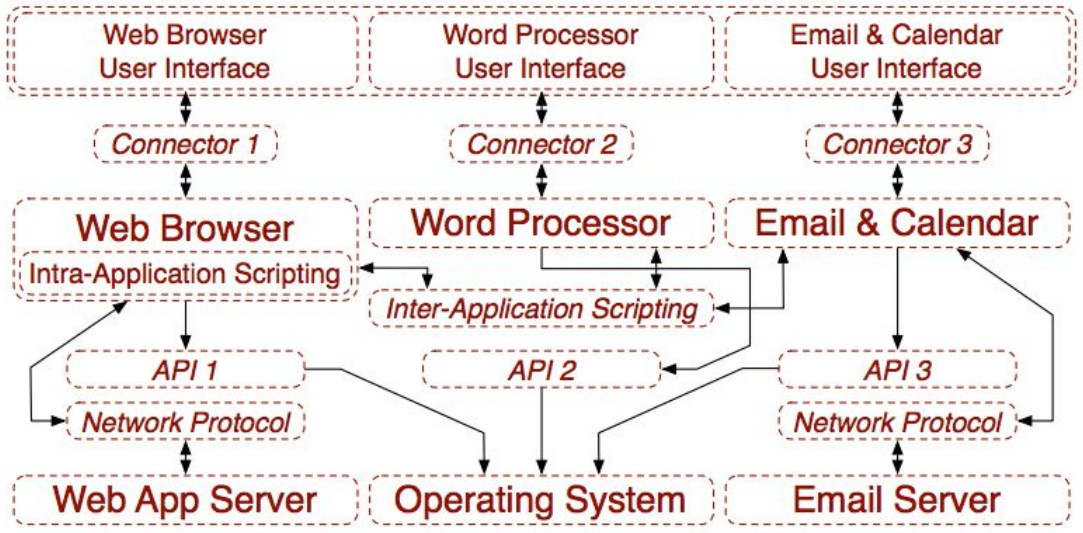


Figure 3: The design-time architecture of the system of Figure 2

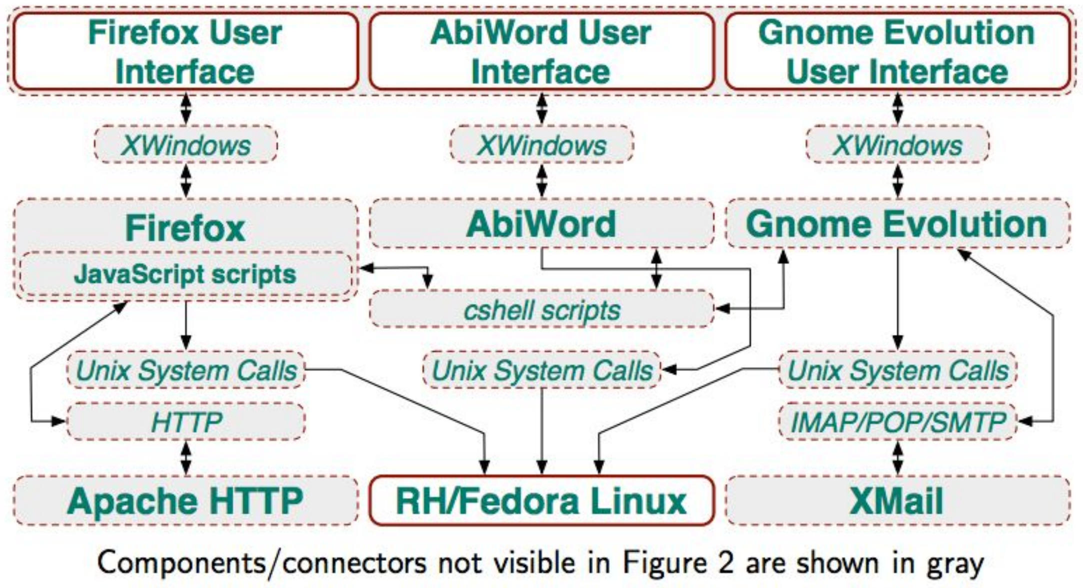


Figure 4: A build-time architecture describing the version running in Figure 2.

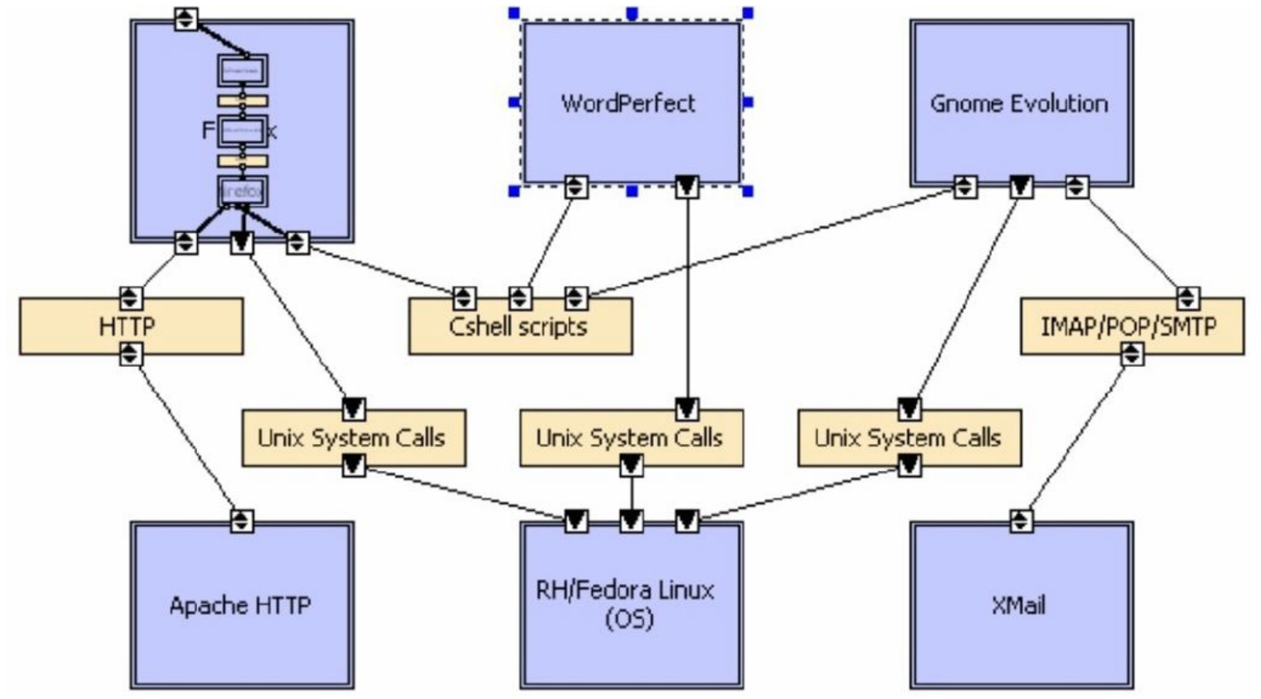


Figure 5. An example OA system modeled using the UCI ArchStudio software architecture development environment [ISR06], corresponding to Figures 2 and 4.

The topology of the build-time architecture also determines the OA software ecosystem of the system, with its dependecies 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 may 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.

OA System 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 during to cybersecurity attacks, or weaknesses in OA system development and acquisition processes.

At the highest level of granularity, a system developer or integrator can choose among alternative producers of similar components, services, and platforms [SWZ12]: For example, we can find *functionally similar* alternatives from software (component) producers of web browsers like Mozilla (Firefox, Camino, Sea Monkey) vs. Google (Chrome) vs. Microsoft (Internet Explorer), vs. others. Similarly, for word processors, we find alternatives including Microsoft (Word) vs. abisoft.com (AbiWord) vs. Google (Google Docs, which is a remote Web service rather than a component), vs. 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 to modify, for example, OA system topology, 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* [NaS87, Bos06] or a *product line* [CN01].

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 [Fra10, SJW11]. Last, software product lines can be bound to a network of software producers, system integrators, and system users/consumers through a software ecosystem [Bos09], such that secure systems can be realized through composition or configuration at the software ecosystem level [ScA12]. 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 to address the challenges of creating secure OA systems through secured software product lines. Secure OA system concepts are presented in a later chapter in this book.

How OA software systems evolve

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

- 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; and
- 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.

The success of DoD's OA and OSS programs in achieving the positive qualities associated with OSS depend on the socio-technical context in which a system is developed and used. The stakeholders and users of an OSS system typically include the developers of that system; they know its goals and requirements implicitly, and can adapt and evolve the system to follow their understanding of the context in which it is used. If DoD is to achieve quick response, rapid adaptation, and context-appropriate use of OSS, it may be necessary to have a representative group of the personnel that are to use and adapt it to the needs they see around them, be OSS developers for that system.

Following from our analysis above, it appears there are a new set of requirements that are emerging that will need to be addressed in any acquisition of a software-intensive system that is stipulated to employ an OA that accommodates OSS components or connectors. Identifying specific requirements for a given program acquisition or system development contract can benefit from consideration of the the following guidelines for how best to realize an OA:

- Determining how much openness is required or desired in an OA software system.
- Identifying guidelines and incentives for software development contractors that encourage them to develop, provide, and distribute/deploy OA systems with OSS components, connectors, and configuration that minimize conflicting OSS license obligations.

- Determining the restrictions, if any, that the OSS licenses used by different software system components, connectors, or configurations within a OA system.
- Identifying alternative OSS component, connector, or configuration candidates that may satisfy a specified overall system architecture.
- Determining scenarios that help reveal whether there are OSS licensing conflicts for a given set of OSS components, connectors, or configuration.
- Identifying and analyzing any OSS licensing obligations that must be satisfied for the resulting system to be available for redistribution.
- Identifying and validating OSS license conformance criteria for configured systems intended for redistribution.

Further elaboration on these guidelines is subject to additional research, application, and refinement. However, they do provide a useful starting point for discussion, debate, and action in program acquisition.

Conclusions

This chapter explored and described open archtiecture techniques for specifying and modeling OA software systems. OA systems are composed and configures using different types of software elements, including components (modules) with explicit interfaces, and connectors that configure systems, sub-systems, or system of systems, through component-interface interconnections. The chapter also described how OA software systems can form software product lines utilizing different selections of specified component types, which is a central technique for realizing software reuse, improving quality (through use of known, stable, and robust software elements), and thus for helping to reduce the cost of OA software system acquisition. Such an approach thus serves to provide a strong foundation for aligning OA concepts that are central to Better Buying Power acquisition initiatives, with other advantages that enable software component and OA reuse across diverse platforms and application domains.

References

[BCK03] Bass, L., Clements, P., and Kazman, R., (2003). Software Architecture in Practice, 2nd Edition, Addison-Wesley Professional, New York.

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

[Bo09] Bosch, J., (2009). From software product lines to software ecosystems. In: *Proc. 13th Intern. Software Product Line Conference* (SPLC'09), 111-119.

[ClN01] Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns. Addison-Wesley, New York.*

[CFG13] Chudnovsky, O., Fischer, C. Gaedke, M. and Pietschmann (2013). Inter-Widget Communication by Demonstration in User Interface Mashups. *Web Engineering*, Springer-Verlag, Lecture Notes in Computer Science, Vol. 7977, 502-505.

[End13] Endres-Niggemeyer, B. (2013). The Mashup Ecosystem, in *Semantic Mashups: Intelligence Reuse of Web Resources*, Springer, 1-50.

[End13a] Endres-Niggemeyer, B. (2013). Mashups Live on Standards, in *Semantic Mashups: Intelligence Reuse of Web Resources*, Springer, 51-89.

[GGM14] George, A., Galdorisi, G., Morris, M. and O'Neil (2014). DoD Application Store: Enabling C2 Agility. *Proc. 19th Intern. Command and Control Research and Technology Symposium* (ICCRTS), Paper-104, Fairfax, VA, June 2014.

[FiT02] Fielding, R. and Taylor, R.N., (2002). Principled Design of the Modern Web Architecture, ACM Transactions Internet Technology, 2(2), 115-150.

[Fel07] Feldt, K., (2007). Programming Firefox: Building Rich Internet Applications with XUL, O'Reilly Press, Sebastopol, CA.

[Fra10] Franz, M. (2010). E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism, *New Security Paradigms Workshop (NSPW'10)*, Sept. 21–23, Concord, Massachusetts, USA.

[ISR06] Institute for Software Research (2006). *ArchStudio 4*. Technical report, University of California, Irvine. http://www.isr.uci.edu/projects/archstudio/

[KWD00] Kuhl, F., Weatherly, R., and Dahmann, J. (2000). Creating Computer Simulation Systems: An Introduction to the High Level Architecture, Prentice-Hall PTR, Upper Saddle River, New Jersey.

[NaS87] Narayanaswamy, K. and Scacchi, W. (1987) Maintaining Configurations of Evolving Software Systems, *IEEE Trans. Software Engineering*, 13(4), 323-334.

[NeC06] Nelson L. and Churchill, E.F., (2006). Repurposing: Techniques for Reuse and Integration of Interactive Services, *Proc. 2006 IEEE Intern. Conf. Information Reuse and Integration*, 490-495, September.

[Ore00] Oreizy, P. (2000). *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution*. PhD thesis, University of California, Irvine. http://www.ics.uci.edu/~peymano/papers/thesis.pdf .

[Ros05] Rosen, L. (2005). Open Source Licensing: Software Freedom and Intellectual Property Law, Prentice-Hall PTR, Upper Saddle River, New Jersey. http://www.rosenlaw.com/oslbook.htm

[SJW11] Salamat, B., Jackson, T., Wagner, G., Wimmer, C., Franz, M. (2011). Run-Time Defense against Code Injection Attacks using Replicated Execution, *IEEE Transactions on Dependable and Secure Computing, Volume 8, No. 4*, July 2011.

[ScA08] Scacchi, W. and Alspaugh, T.A. (2008). Emerging Issues in the Acquisitio of Open Source Software within the Department of Defense, *Proc. 5th. Annual Acquisition Research Symposium*, Vol. 1, 230-244, NPS-AM-08-036, Naval Postgraduate School, Monterey, CA.

[ScA12] Scacchi, W. and Alspaugh, T.A. (2012). Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems, *Journal of Systems and Software*, 85(7), 1479-1494, July 2012.

[ScA13b] Scacchi, W. and Alspaugh, T.A. (2013b). Streamlining the Process of Acquiring Secure Open Architecture Software Systems, *Proc. 10th Annual Acquisition Research Symposium*, Monterey, CA, 608-623, May 2013.

[StL04] St. Laurent, A.M., (2004). Understanding Open Source and Free Software Licensing, O'Reilly Press, Sebastopol, CA.

[SWZ12] Sun, K., Wang,J., Zhang, F. and Stavrou, A. (2012). SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes. *Proc. 19th. Annual Network and Distributed System Security Symposium.*

[UnT08] Unity Technologies, Dec. 2008. End User License Agreement. http://unity3d.com/unity/unityend-user-license-2.x.html.

Chapter 3.

License Challenges for Open Architectures

Chapter 3.

License Challenges for Open Architectures

Abstract

An open architecture (OA) is typically instantiated using components of different intellectual property (IP) licenses. Interactions between the licenses pose challenges that do not occur in older types of systems, in which all parts of the system are governed by a single license. This chapter identifies and examines a number of these challenges.

Introduction

It is quite possible to design an OA software system whose component licenses interact in undesirable ways, so that in the extreme case the system cannot legally be used (for example, a system that is a work based on both GPLv2-licensed components and proprietary components). Less extreme consequences are also possible, such as:

- A system without desired rights, such as one that can be used, but not distributed (see ***A Multimedia Content Management Portal***, below);
- A system that has less that optimal capabilities, because inferior components were used in order to simplify the license challenges (for example, an OA system developed without any GPL-licensed components);
- A system that can involve its users and developers in undesired license obligations (see ***Bank of America and Merrill Lynch***, below);
- Or a system whose license rights and obligations cannot be determined by its potential adopters (see ***Unity***, below and in Chapter 6 on ecosystems);

Until recently, the norm for licensed software has been that software is used and distributed under the terms of a single license, with all its components homogeneously licensed under a single proprietary or open-source software (OSS) license. It is increasingly common to see heterogeneously-licensed (HtL) systems, whose components are not under the same license [GeH09, ScA08, UEULA08]. For web systems this has become so common that commercial tools for creating such “mashups” have been available for several years [NeC06, End13]. Carefully constrained design, possibly aided by license exceptions from the copyright owners, may enable the resulting system to have a single specific license [GeH09]. Otherwise the system as a whole has no single license, but rather one or more rights that are the intersection of all the component license’s rights, and the union of their obligations. An example of a HtL system is the Unity game development tool, whose license agreement lists eleven distinct licenses for its components, in addition to its overall license terms granting the right to use the system [UEULA08].

The hallmark of Free/Open Source Software (FOSS) is that the source code is available for remote access, open to study and modification, and available for redistribution to others with few constraints, except the rights and obligations that insure these freedoms. FOSS sometimes adds or removes similar freedoms or copyright privileges depending on which FOSS copyright and end-user license agreement is associated with a particular FOSS code base [FKM08, Ros05]. Some licenses for “free software” such as the group of GNU General Public License (GPL) and Lesser General Public License (LGPL) versions [FSF91, FSF99, FSFAGPL07, FSFGPL07, FSFLGPL07] are well known and widely used, while others are obscure and sometimes specific to a particular software vendor. The Open Source Initiative [OSI15], whose Web portal gives information on many facets of FOSS, especially FOSS licenses, currently certifies more than 50 FOSS licenses, thus indicating a diverse ecology of freedoms, copying rights, and other license obligations and constraints.

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

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

Some FOSS licenses overlap or subsume one another's rights, while others present potential conflicts when comparing one license to another. Consequently, FOSS developers generally choose a single license to apply to their FOSS project, as part of their governance regime [deL07]. The choice of FOSS license to apply has been a defining characteristic of most FOSS projects, where the license chosen may connote not only an intellectual property sharing regime, but also a statement about beliefs, values, and norms expected to be shared by FOSS project developers, as well as affiliation within a larger social movement [EIS03, EIS08, RHS06]. However, a single license may not be sufficient to provide “copyleft” access to non-software specific data objects, representations, processing rules, or visual renderings. Similarly, with ever more FOSS components becoming available with different FOSS licenses, and some now even offered under multiple licenses or recognizing that different versions of a given software component may have different licenses or license constraints, then software and information system developers face a growing challenge: to determine how multiple software licenses interact, whether during system design (i.e. at “design time”), while compiling and linking source code to produce an executable program/binary (at “build time”), or when installing and running a newly acquired/downloaded version of software from a FOSS project or other provider that may need to interoperate with other software programs (at “run time”).

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

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

The fundamental problem we must address is to understand and analyze what happens when software systems are developed from FOSS or proprietary components that are not all under the same license. What license applies to the resulting system? What rights or obligations apply? How can one determine which license constraints match, subsume, or conflict with one another? We refer to this problem as the challenge of heterogeneously-licensed systems (HLSs), and we find that a growing number of firms and government agencies must increasingly address this challenge.

A further problem 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

Examples

Bank of America and Merrill Lynch

When the Bank of America took over operation of the Merrill Lynch (ML) trading firm in 2008, ML was known as a leading developer of in-house financial and trading systems incorporating FOSS components. However, the Bank now had to rapidly determine whether this corporate takeover constituted a “software distribution” by ML, in which case the source code for the ML software would have to be made publicly available, as well as what consequences might arise from integrating the ML systems with its own [Asa08], as this would likely create an HLS.

Unity

Unity3D is an interactive environment for modeling and animating 3D graphic objects and object compositions within computer games or virtual worlds. Unity3D is an HLS, as seen in its software copyright license agreement that lists 18 externally produced components or groups of components, apparently distributed under eleven distinct licenses [UEULA08]. So what “license” rights and obligations apply to this software? Is it the concatenation, union, or intersection of the license constraints found in each of the identified license copyrights? Do any of these license constraints conflict with one another (e.g. stipulating no redistribution of software, versus ensuring the right to software redistribution)? How does the architectural configuration of the software components associated with a given license affect which component licenses interact, and which do not? Understanding the rights and obligations incorporated into the Unity3D system license requires understanding and analyzing the corresponding terms and conditions of each of these licenses, and potentially understanding the architecture in which Unity3D incorporates them. This is the burden facing software consumers, and it appears to be one that is growing. It is also a burden facing software integrators, as they must ensure they can give appropriate rights (and impose acceptable obligations) for their consumers.

The current state of the art of research in the fields of law and FOSS does not address these concerns, as we will show in more detail in the Related Research section below. Legal researchers have examined interactions between pairs of FOSS licenses in the abstract, but not in the context of real systems and the architectural components and configurations found there. FOSS research in this area has concentrated on recovering or confirming the license of an individual homogeneously-licensed software component, with only one group (other than ourselves) examining the application of FOSS licenses in the context of actual HLSs, and that only of pairs of licenses applied in a small fixed number of architectural contexts. None of these approaches provide answers to the questions posed above.

A Multimedia Content Management Portal

Shortly before beginning the research described here, we prototyped a new multimedia content management portal that included support for videoconferencing and video recording and publishing. Our prototype was based on an existing Adobe Flash Media Server (FMS), for which we developed both broadcast and multi-cast clients for video and audio that shared their data streams through the FMS. FMS is a closed source media server for which the number of concurrent client connections is limited, with the limit determined by the license fee paid. We could invite remote users to try out our clients and media services (up to the connection limit), but since the FMS license did not allow redistribution, we found we could not offer interested users a copy of the run-time environment that included the FMS. We could distribute everything but the FMS, though our compiled components were built to use our copy of the FMS. Consequently, recipients would be unable to run the system even if they purchased their own FMS license. The only useful way to distribute our portal system was in the form of the source code of our locally-developed clients and services. A potential user would need to license, download, and install his own copy of the FMS, configure our source code to use it, and rebuild our system. In our view, this created a barrier to sharing the emerging results of our prototyping effort.

We subsequently undertook to replace the FMS with Red5, an open source Flash media server, so we could distribute a complete run-time version of our content management portal to remote developers. Now these developers could install and use our run-time system as-is, or if they preferred they could download the source code, revise, build, and configure it to suit their own needs, and share their own run-time version.

Our experience illustrates how heterogeneous licensing can interact with system architecture decisions to hamper common software research and development activities in surprising ways, even for experienced FOSS designers and developers, and if not planned for successfully can cause substantial unexpected costs and delays. Our license theory, embodied in the tool described later in the paper, would have highlighted the lack of distribution rights and modeled the non-substitutability of the FMS server at system design time rather than much later at system distribution and run time.

Related Research

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 FOSS 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 FOSS license for use in a FOSS development project, and its consequences in determining the likely success of such a project.

Brown and Booch discuss issues that arise in the reuse of FOSS 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 FOSS-based systems [BrB02]. 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.

Stewart et al. conducted an empirical study to examine whether license choice is related to FOSS project success, finding a positive association following from the selection of business-friendly licenses [SAM06]. Sen, Subramaniam, and Nelson in a series of studies [Sen07, SSN08, SSN09] similarly find positive relationships between the choice of a FOSS license and the likelihood of both successful FOSS development and adoption of corresponding FOSS systems within enterprises. These studies direct attention to FOSS projects that adopt and identify their development efforts through use of a single FOSS 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 FOSS and proprietary licenses may be involved.

Legal scholars have examined FOSS licenses and how they interact in the legal domain, but not in the context of HLSs. St. Laurent examines twelve FOSS licenses, including several no longer in wide use, and compares them to a hypothetical proprietary license he created; license interactions and conflicts are only very briefly discussed, and only in general terms [StL04]. Rosen surveys eight FOSS licenses and creates two new ones written to professional legal standards [Ros05]. He examines interactions primarily in terms of the general categories of reciprocal and non-reciprocal licenses, rather than in terms of specific licenses. Rosen was general counsel for the Open Source Initiative [OSI15]. Fontana et al. primarily focus on guidance for FOSS projects on a number of legal issues, but provides a good and authoritative survey of FOSS licenses, especially the GPL group of licenses [FKM08]. Fontana et al. are lawyers (with one exception) associated with the Software Freedom Law Center; Fontana and Moglen were two of the authors of the GPL, LGPL, and AGPL version 3 licenses. Finally, Kemp reviews significant court cases that have been pursued, and how they do or do not address issues concerning the propagation of rights and obligations across programs depending on how they are derived, contained, compiled, or linked at build time, especially when the GPLv2 license is involved [Kem09]. Common to this legal scholarship is an approach that analyzes licenses and the interactions among them abstractly rather than in the context of an HLS, and on at most a pairwise basis. The characteristics of the software in which the licenses interact are not taken into account, or at most in very general terms, even though almost every FOSS license is framed in terms of the software and architectural constructs in existence when the license was written.

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 and Mannaert are a recent exception.

Ven and Mannaert discuss the challenges faced by independent software vendors developing an HLS. They focus on the evolution and maintenance of modified FOSS components [VeM08]. Tuunanen et al. exemplify most work to date on HLSs, in that they focus on reverse engineering and recovery of individual component licenses on existing systems, rather than on guiding HLS design to achieve and verify desired license outcomes [TKK09]. Their approach does not support the calculation of HLS virtual licenses. Many more researchers have worked from this after-the-fact point of view [Gob08, DGG10].

Discussion

Software systems with open architectures are subject to more and 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. [OSI15]). Software components are now available subject to licenses such as the General Public License (GPL), Mozilla Public License (MPL), Apache Public License, (APL), Academic licenses (e.g., BSD, MIT), Creative Commons, Artistic, and others as well as Public Domain (either via explicit declaration or by expiration of prior copyright license). Furthermore, licenses such as these can evolve, resulting in new license versions over time. But no matter their diversity, software licenses represent a legally enforceable contract that is recognized by government agencies, corporate enterprises, individuals, and judicial courts, and thus they cannot be taken trivially. As a consequence, software licenses constrain open architectures, and thus architectural design decisions.

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) the fact that many alternative architectural configurations and APIs exist that may or may not produce an OA (cf. [AIA07, ScA08]). Subsequently, we believe this can lead to situations in which new software development or acquisition requirements stipulate a software system with an OA and OSS, but the resulting software system may or may not embody an OA. This can occur when the architectural design of a system constrains system requirements—raising the question of what requirements can be satisfied by a given system architecture, when requirements stipulate specific types or instances of OSS (e.g., Web browsers, content management servers) to be employed [Sca02], or what architecture style [BCK03] 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 [ASA10, ScA08].

Conclusions

The interactions between open architecture, open source software, and software licenses are poorly understood. Architectures that show no obvious warning signs of possible trouble ahead nevertheless can result in serious problems down the road, as the examples discussed in this chapter illustrate. Given the goal of realizing an OA strategy using OSS components and open APIs, it has been unclear how best to align software architecture decisions, OSS component choices, and software license regimes to achieve this goal. Subsequent chapters examine particular issues and viewpoints, and offer approaches for addressing these challenges.

References

[AlA07] Alspaugh, T.A and Antón, A.I., (2007). Scenario Support for Effective Requirements, *Information and Software Technology*, 50(3), 198-220.

[ASA10] Alspaugh, T.A., Scacchi, W. and Asuncion, H.A. (2010). Software Licenses n Context: The Challenges of Heterogeneously-Licensed Systems,

[Asa08] Asay, M. (2008). In Acquiring Merrill Lynch, must Bank of America Open Source Its Software? *CNET News*, http://news.cnet.com/8301-13505_3-10043029-16.html, 16 September 2008.

[BCK03] Bass, L., Clements, P., and Kazman, R., (2003). *Software Architecture in Practice*, 2nd Edition, Addison-Wesley Professional, New York.

[BrB02] Brown, A.W. and Booch, G. (2002). Reusing open-source software and practices: The impact of open-source on commercial vendors. In *Proc. 7th. Intern. Conf. Software Reuse: Methods, Techniques, and Tools (ICSR-7)*, 123-136, April.

[deL07] de Laat, P.B. (2007). Governance of Open Source Software: State of the Art, *J. Management and Governance*, 11(2), 165–177.

[DGG10] Di Penta, M., German, D. M., Gueheneuc, Y.-G., and Antoniol, G. (2010). An Exploratory Study of the Evolution of Software Licensing. *29th International Conference on Software Engineering,* (ICSE ‘10), 145-154.

[ElS03] Elliott, M. S., and Scacchi, W. (2003). Free Software Developers as an Occupational Community: Resolving Conflicts and Fostering Collaboration. *ACM International Conference on Supporting Group Work* (GROUP’03), 21-30.

[ElS08] Elliott, M. S., and Scacchi, W. (2008). Mobilization of Software Developers: The Free Software Movement. *Information Technology & People*, 21(1):4–33.

[End13] Endres-Niggemeyer, B. (2013). The Mashup Ecosystem, in *Semantic Mashups: Intelligence Reuse of Web Resources*, Springer, 1-50.

[FKM08] Fontana, R., Kuhn, B.M., Moglen, E. M. *et al*. (2008). *A Legal Issues Primer for Open Source and Free Software Projects*. Software Freedom Law Center.

[FSF91] Free Software Foundation (1991). *GNU General Public License Version 2*. http://opensource.org/licenses/gpl-2.0.php

[FSF99] Free Software Foundation (1999). *GNU Lesser General Public License Version 2.1*. http://opensource.org/licenses/lgpl-2.1.php

[FSFAGPL07] Free Software Foundation (2007a). *GNU Affero General Public License Version 3*. http://opensource.org/licenses/agpl-v3.html

[FSFGPL07] Free Software Foundation (2007b). *GNU General Public License Version 3*. http://opensource.org/licenses/gpl-3.0.html

[FSFLGPL07] Free Software Foundation (2007c). *GNU Lesser General Public License Version 3*. http://opensource.org/licenses/lgpl-3.0.html

[GeH09] German, D.M. and Hassan, A.E. (2009). License integration patterns: Dealing with licenses mis- matches in component-based development. In *Proc. 28th International Conference on Software Engineering (ICSE'09)*, May 2009.

[Gob08] Gobeille, R. (2008). The FOSSology project. In *International Working Conference on Mining Software Repositories* (MSR'08):47–50.

[NeC06] Nelson L. and Churchill, E.F., (2006). Repurposing: Techniques for Reuse and Integration of
Interactive Services, *Proc. 2006 IEEE Intern. Conf. Information Reuse and Integration*, 490-495, September.

[Kem09] Kemp, R. (2009). Current Developments in Open Source Software. *Computer Law and Security Review*, 25(6):569–582.

[OSI15] OSI (2015). *Open Source Initiative*. http://www.opensource.org/

[RHS06] Roberts, J. A., Hann, I.-H., and Slaughter, S. A. (2006). Understanding the Motivations, Participation, and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects. *Management Science*, 52(7):984–999.

[Ros05] Rosen, L. (2005). *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall.

[Sca02] Scacchi, W. (2002). Understanding the Requirements for Developing Open Source Softrware Systems, *IEE Proceedings--Software*, 149(1), 24-39, February 2002.

[ScA08] Scacchi, W. and Alspaugh, T.A. (2008). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proc. 5th Annual Acquisition Research Symposium*, Monterey, CA.

[Sen07] Sen, R. (2007). A strategic analysis of competition between open source and proprietary software. *Journal of Management Information Systems*, 24(1), 233–257, 2007.

[SSN09] Sen, R., Subramaniam, C. and Nelson, M.L.. (2009) Determinants of the choice of open source software license. *Journal of Management Information Systems*, 25(3), 207–240.

[SAM06] Stewart, K.J., Ammeter, A.P., and Maruping, L.M. (2006). Impacts of license choice and organiza- tional sponsorship on user interest and development activity in open source software projects. *Information Systems Research,* 17(2), 126–144, 2006.

[SSN09] Subramaniam, C., Sen, R. and Nelson, M.L. (2009). Determinants of open source software project success: A longitudinal study. *Decision Support Systems*, 46(2), 576–585.

[StL04] St. Laurent, A.M. (2004). *Understanding Open Source and Free Software Licensing*. O'Reilly Media, Inc., Sebastopol, CA.

[TKK09] Tuunanen, T., Koskinen, J., and Karkkainen, T. (2009). Automated software license analysis. *Automated Software Engineering*, 16(3-4), 455–490.

[UEULA08] Unity Technologies. *End User License Agreement*, Dec. 2008. http://unity3d.com/unity/unity- end-userlicense-2.x.html

[VeM08] Ven, K. and Mannaert, H. (2008). Challenges and strategies in the use of open source software by independent software vendors. *Information and Software Technology*, 50(9-10), 991–1002.

Chapter 4.

Software License Legal Foundations

Chapter 4.

Software License Legal Foundations

Abstract

Software is commonly treated as a kind of intellectual property (IP) that may be protected from infringing uses through software licenses. Such licenses depend primarily on copyright law as the basis for controlling what can be done with the licensed software, with patent and trademark law also used in some licenses. Licenses grant rights desired by licensees in exchange for obligations that address the goals of the licensors. For proprietary licenses, these goals are usually financial gain, protection of the licensor's exclusive rights in the software, and limitations on the licensor's liability. In contrast, for open source software (OSS) a primary goal is the broad and free distribution of the software and its development artifacts, and in many cases the permanent protection of that distribution. The rights can be traced back to the exclusive rights defined by copyright, patent, and trademark law, and for OSS licenses important obligations can be as well. We use the actions that rights and obligations refer to as the key concept, and identify subsumption relationships among rights and obligations through the subsumption among the actions they involve. This relationship among exclusive rights established by law, license rights, and license obligations makes it possible to create lasting ecosystems of open software and to combine software modules governed by various licenses in open architectures.

License Structure

Under U.S. law and the law of most countries, a license can be either a bare license or a contract license. A bare license simply grants one or more copyright or patent rights from the copyright and/or patent holder to another person, typically under specified conditions. A contract license is constructed in the form of a contract, involving an exchange of promises between the parties. In a contract license, the licensor grants one or more rights in exchange for some consideration from the licensee receiving them. The consideration given by the licensee may be very small, as little as "a peppercorn" in the traditional explanation, but it cannot be nothing. However, in addition to a payment or other obvious consideration, it can take the form of "(a) an act other than a promise, or (b) a forbearance, or (c) the creation, modification, or destruction of a legal relation", which are an expression of conditions in the common law of torts and contracts [ALI81]. In FOSS licensing the fundamental consideration is typically interpreted as the licensee's "detrimental reliance" on the licensed rights, or in other words the reliance on the software that would be to the licensee's detriment if the software were withdrawn. A license, whether bare or contract, can also impose specific non-consideration obligations as a condition of the license grant. Most FOSS licenses are drawn as contract licenses in order to benefit from the well established case law on interpretation of contract provisions, with the exception of the GPL family of licenses which are drawn as bare licenses [Ros05, Gor89, Det06, Gua09, HiO09, Kem09].

Intellectual Property

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

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

Software licenses are primarily concerned with copyrights and patents, and mention trademarks only to restrict a licensee's use of them; licenses rarely discuss trade dress or trade secrets [Ros05]. In this chapter we focus on copyright aspects of licenses.

Copyright is defined by Title 17 of the U.S. Code and by similar law in many other countries. It grants exclusive rights to the author of an original work in any tangible means of expression, namely the rights to

- reproduce the copyrighted work;
- distribute copies;
- prepare derivative works;
- distribute copies of derivative works; and
- (for certain kinds of work) perform or display it.

Because the rights are exclusive, the author can prevent others from exercising them, except as allowed by "fair use". The author can also grant others any or all of the rights or any part of them; one of the functions of a software license is to grant such rights, and define the conditions under which they are granted [USC11].

Copyright subsists in the expression of the original work, that is, the rights begin from the moment the work is expressed. In the U.S. a copyright lasts for the author's lifetime plus 70 years, or 95 years for works for hire [USC11].

Software Licenses

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

Permissive OSS licenses such as the Berkeley Software Distribution (BSD) license, the Apache Software License, and perl's Artistic License [Als15] grant nearly all rights and impose few obligations. Typical permissive license obligations are simply to not remove the copyright and license notices.

Reciprocal (also known as "copyleft") OSS licenses impose an obligation that distributed modifications of reciprocally-licensed software be freely licensed under the same license. Examples are the Lesser General Public License (LGPL), Mozilla Public License (MPL), and Common Public License [Als15].

Some reciprocal licenses additionally require that software combined with the licensed software (for various definitions of "combined") also be freely licensed under the same license. We term such reciprocal licenses *propagating*; they are also known as "strong copyleft" licenses. Examples are the General Public License versions 2 and 3 (GPLv2, GPLv3) [Als15].

OSS licenses typically disclaim liability, assert that no warranty is implied, and obligate licensees to not use the licensor's name or trademark.

Newer licenses often cover patent issues as well and the rights to make, use, sell or offer for sale, and import that are governed by patent law. These licenses either grant a restricted patent license or explicitly exclude the granting of patent rights. However, some important licenses are constructed so that some or all rights under the license terminate if the licensee institutes patent infringement suits related to the licensed software (specifics vary by license), for example Apache 2.0 and MPL 1.1. Proprietary licenses often place limits on the use of the licensed software as part of the contractual obligations imposed by the license, whether the software is patented or not.

Several newer licenses add interesting degrees of flexibility. Most licenses grant the right to sublicense under the same license, or in some cases under any version of the same license. IBM's CPL grants the right to sublicense under any license that meets certain conditions; CPL itself meets them, of course, but several other licenses do as well.

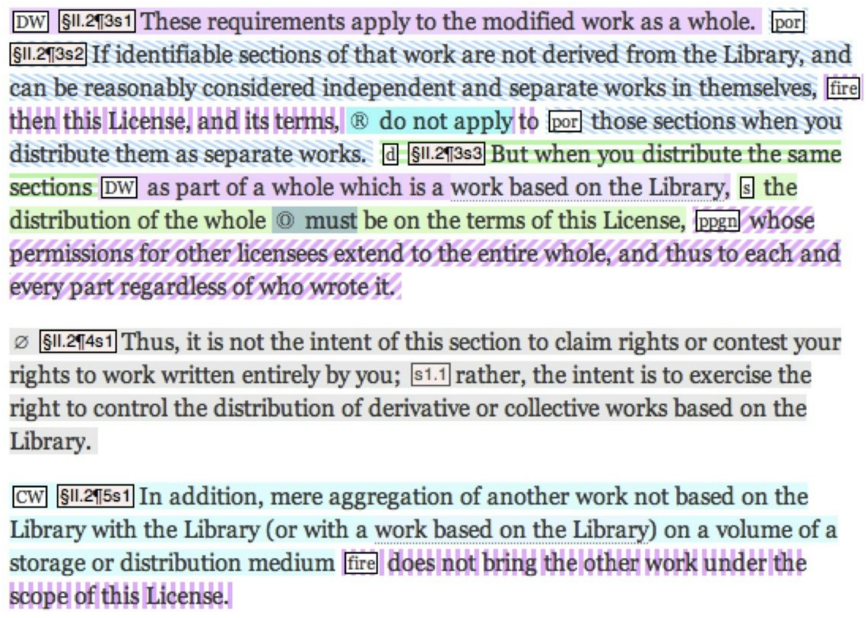
The Open Source Initiative (OSI) maintains standards for OSS licenses, reviews OSS licenses under those standards, and gives its approval to those that meet them [OSI15]. OSI publishes a standard repository of approximately 70 approved OSS licenses.

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

What Licenses Contain

In this section we focus on the Lesser General Public License (LGPL), version 2.1 [FSF99, Als15]. LGPLv2.1 is the seventh most widely used open-source software (OSS) license, accounting for about 6.5% of open source projects [BDS12]. At 4341 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:

- 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.


DW §II.2¶3s1 These requirements apply to the modified work as a whole. por
§II.2¶3s2 If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, fire then this License, and its terms, ® do not apply to por those sections when you distribute them as separate works. d §II.2¶3s3 But when you distribute the same sections DW as part of a whole which is a work based on the Library, s the distribution of the whole Ⓞ must be on the terms of this License, ppgn whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

∅ §II.2¶4s1 Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; s1.1 rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

CW §II.2¶5s1 In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium fire does not bring the other work under the scope of this License.

Figure 1. A portion of LGPLv2.1, divided into chunks and annotated with categories of interest. 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.

Figure 1 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 [CoS07]. 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 section “Textual Analysis of a Specific License”, and the characterization of a parameterized action as the basic unit of software licenses discussed in in the section “Actions, The Central Construct”.

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 [Als15].

Textual Analysis of a Specific License

We find that everything in the text of LGPLv2.1 may be classified as either

1. the definition of a term,
2. a right,
3. an obligation,
4. a modifier to a definition, right, or obligation, or
5. text without legal effect.

These five categories cover the entire text and partition everything in it. Examples of each from LGPLv2.1 are given below.

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”. (§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”. 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 below is an explanation and statement of the intent of the license's authors; however in law, we are told, 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 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 [Hoh13], 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 will be discussed below.

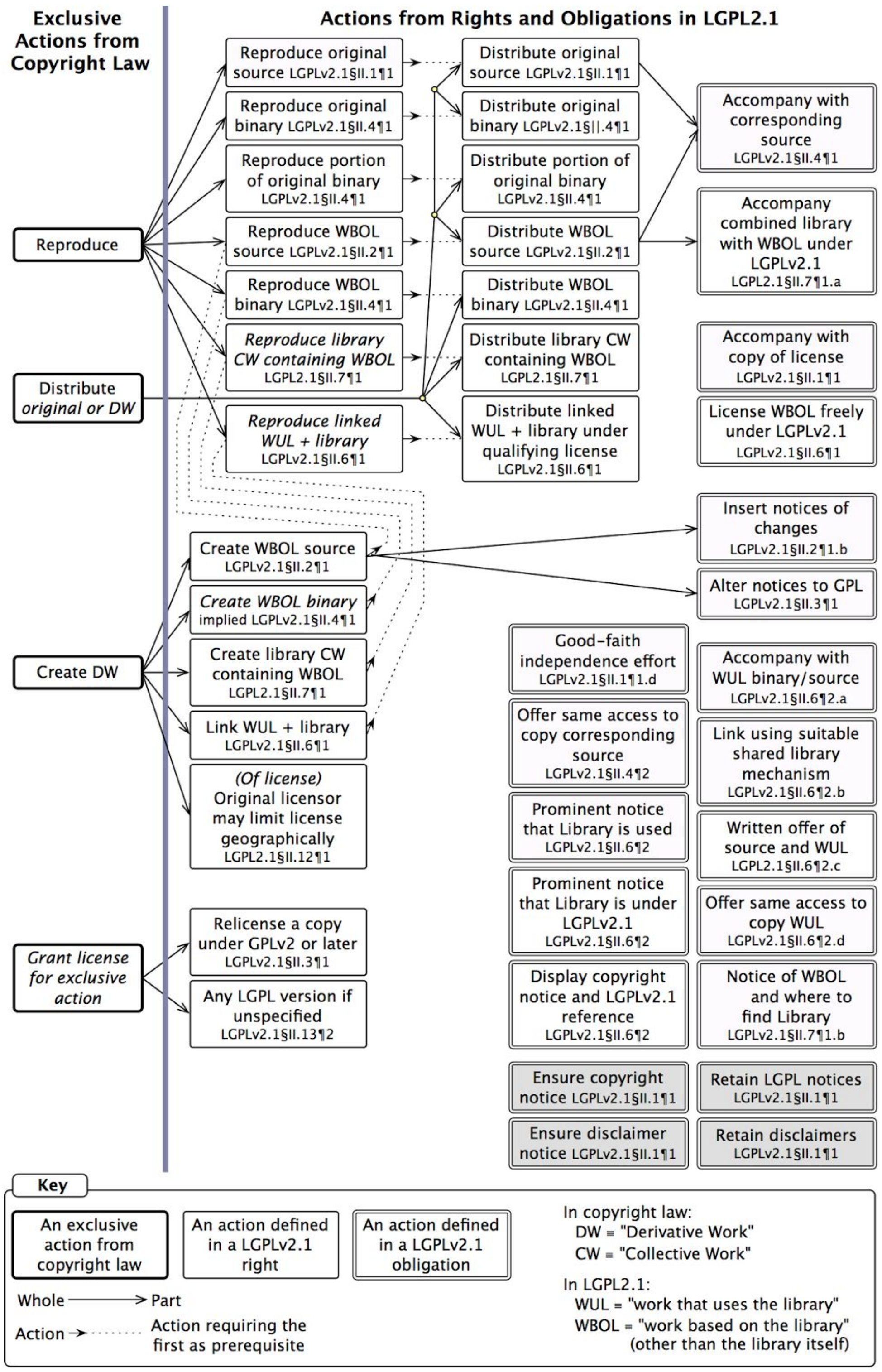

Exclusive Actions from Copyright Law
Actions from Rights and Obligations in LGPL2.1
Reproduce
Distribute original or DW
Create DW
Grant license for exclusive action
Reproduce original source LGPLv2.1§II.1¶1
Reproduce original binary LGPLv2.1§II.4¶1
Reproduce portion of original binary LGPLv2.1§II.4¶1
Reproduce WBOL source LGPLv2.1§II.2¶1
Reproduce WBOL binary LGPLv2.1§II.4¶1
Reproduce library CW containing WBOL LGPL2.1§II.7¶1
Reproduce linked WUL + library LGPLv2.1§II.6¶1
Distribute original source LGPLv2.1§II.1¶1
Distribute original binary LGPLv2.1§||.4¶1
Distribute portion of original binary LGPLv2.1§II.4¶1
Distribute WBOL source LGPLv2.1§II.2¶1
Distribute WBOL binary LGPLv2.1§II.4¶1
Distribute library CW containing WBOL LGPL2.1§II.7¶1
Distribute linked WUL + library under qualifying license LGPLv2.1§II.6¶1
Accompany with corresponding source LGPLv2.1§II.4¶1
Accompany combined library with WBOL under LGPLv2.1 LGPL2.1§II.7¶1.a
Accompany with copy of license LGPLv2.1§II.1¶1
License WBOL freely under LGPLv2.1 LGPLv2.1§II.6¶1
Insert notices of changes LGPLv2.1§II.2¶1.b
Alter notices to GPL LGPLv2.1§II.3¶1
Create WBOL source LGPLv2.1§II.2¶1
Create WBOL binary implied LGPLv2.1§II.4¶1
Create library CW containing WBOL LGPL2.1§II.7¶1
Link WUL + library LGPLv2.1§II.6¶1
(Of license) Original licensor may limit license geographically LGPL2.1§II.12¶1
Relicense a copy under GPLv2 or later LGPLv2.1§II.3¶1
Any LGPL version if unspecified LGPLv2.1§II.13¶2
Good-faith independence effort LGPLv2.1§II.1¶1.d
Offer same access to copy corresponding source LGPLv2.1§II.4¶2
Prominent notice that Library is used LGPLv2.1§II.6¶2
Prominent notice that Library is under LGPLv2.1 LGPLv2.1§II.6¶2
Display copyright notice and LGPLv2.1 reference LGPLv2.1§II.6¶2
Ensure copyright notice LGPLv2.1§II.1¶1
Ensure disclaimer notice LGPLv2.1§II.1¶1
Accompany with WUL binary/source LGPLv2.1§II.6¶2.a
Link using suitable shared library mechanism LGPLv2.1§II.6¶2.b
Written offer of source and WUL LGPL2.1§II.6¶2.c
Offer same access to copy WUL LGPLv2.1§II.6¶2.d
Notice of WBOL and where to find Library LGPLv2.1§II.7¶1.b
Retain LGPL notices LGPLv2.1§II.1¶1
Retain disclaimers LGPLv2.1§II.1¶1
Key
An exclusive action from copyright law
An action defined in a LGPLv2.1 right
An action defined in a LGPLv2.1 obligation
Whole → Part
Action → Action requiring the first as prerequisite
In copyright law:
DW = "Derivative Work"
CW = "Collective Work"
In LGPL2.1:
WUL = "work that uses the library"
WBOL = "work based on the library" (other than the library itself)

Figure 2. Subsumption among the LGPLv2.1 actions for rights and obligations and the exclusive copyright actions. 18 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.

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

2) Time and State: Time and state are barely present in LGPLv2.1, figuring only in license acceptance (§II.9) and termination (§II.8). For example, 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:

1. to other elements incorporated into the same library (§II.2¶1.c);
2. to programs designed to use an LGPLv2.1-licensed library, when linked to the library (§II.5¶2), except when certain conditions are met (§II.6¶1); and
3. 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 above. Every action (even the questionable one) is, unsurprisingly, enactable.

Actions, The Central Construct

Our research has found that license provisions are most effectively represented using a flexibly extensible approach in which the fundamental unit is an action. Rights and obligations then express relationships among desired, required, and forbidden actions. 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. 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 and the Berne convention [USC11], [BCP79]. Where possible, we also identified subsumptions of the actions of license obligations by the actions of rights. Figure 2 shows the subsumption relationships identified for a single license’s actions.

Focusing on actions as the key element of licenses brings several advantages.

- Actions are more manageable than rights and obligations. Each action is a concept representing an unbounded set of instances of the action; e.g. “distribute the Library … in object code … form” (§II.4¶1) is instantiated by “distribute 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 (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 metamodel becomes simpler and more uniform.

Action Parameterization

Here is an example drawn from LGPLv2.1. 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, since 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 section "Parameterized Subsumption".

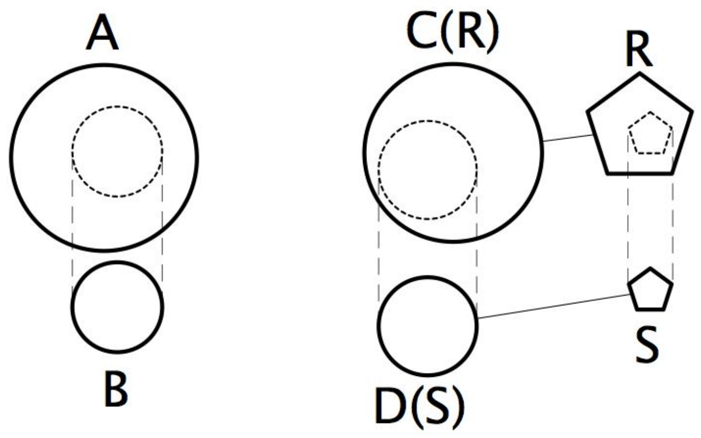


Figure 3. Subsumption of simple entities and parameterized entities

Parameterized Subsumption

In addressing subsumption among parameterized actions, we follow the approach of Abadi and Cardelli in the area of object-oriented type systems [AbC96]. Figure 3 illustrates subsumption between pairs of simple actions and pairs of parameterized actions.

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 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

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

and also assert that

"modify your copy or copies of the Library" (§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 changed the 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.

Conclusions

We present initial results from an analysis of LGPLv2.1 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 2), 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

[AbC96] Abadi, M., and Cardelli, L. (1996). *A Theory of Objects*. Springer-Verlag, New York.

[ALI81] American Law Institute (1981). *The Restatement (Second) of Contracts*.

[Als15] Alspaugh, T.A. (2015). *OSS (and Other) Licenses*, http://www.thomasalspaugh.org/pub/osl-sps/index.html

[BCP79] Berne Convention (1979). *Berne Convention for the Protection of Literary and Artistic Works*, (1979). http://www.wipo.int/treaties/en/ip/berne/

[BDS12] Black Duck Software (2009). *Top 20 Most Commonly Used Licenses in Open Source Projects*. http://www.blackducksoftware.com/oss/licenses

[CoS07] Corbin, J. M. and Strauss, A. C. (2007). *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications.

[Det06] L. Determann. Dangerous liasons—software combinations as derivative works? *Berkeley Technology Law Journal*, 21(4), 2006.

[FSF99] Free Software Foundation (1999). *GNU Lesser General Public License Version 2.1*. http://opensource.org/licenses/lgpl-2.1.php

[Gor89] Gordon, W. J. (1989). An Inquiry into the Merits of Copyright: The Challenges of Consistency, Consent, and Encouragement Theory. *Stanford Law Review*, 41(6):1343–1469.

[Gua09] Guadamuz, A. (2009). The License/Contract Dichotomy in Open Licenses: A Comparative Analysis. *University of La Verne Law Review*, 30(2):101–116.

[HiO09] Hillman, R. A. and O'Rourke, M. A. (2009). Rethinking Consideration in the Electronic Age. *Hastings Law Journal*, 61:311–336.

[Hoh13] W. N. Hohfeld. Some fundamental legal conceptions as applied in judicial reasoning. *Yale Law Journal*, 23(1):16–59, 1913.

[Kem09] Kemp, R. (2009). Current Developments in Open Source Software. *Computer Law and Security Review*, 25(6):569–582.

[OSI15] OSI (2015). *Open Source Initiative*. http://www.opensource.org/

[Ros05] L. Rosen. *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall, 2005.

[USC11] -U.S. Copyright Act, 17 U.S.C. http://www.copyright.gov/title17/

Chapter 5.

Automating License Analysis

Chapter 5.

Automating License Analysis

Abstract

We find that to effectively address the license challenges for open architectures (Chapter 3) it is necessary to do license analysis through automation:

- Identifying the license conflicts, available rights, and corresponding obligations for a proposed system is in general too tedious and too difficult to be done manually with an appropriate level of confidence and accuracy.
- The process of working through an open architecture and potential instantiations of it requires doing a license analysis of every alternative that is seriously considered, quickly enough that the design process can move along at an appropriate speed.
- It is not obvious what characteristics of an architecture, ecosystem, or instantiation of an arbitrary system must be considered in an appropriately thorough license analysis without automating the analysis process.

In this chapter we examine how the process of license analysis can be automated, discuss the more important factors that affect automated license analysis, examine the fundamental structure of licenses and specific parts of some licenses of interest, and discuss an automated analysis procedure and how it can be used to guide system design and instantiation.

Introduction

An increasing number of development organizations are adopting a strategy in which software-intensive systems are composed of heterogeneously licensed (HtL) components, with different components governed by different software licenses. The components are either open source software (OSS) or proprietary software with open application programming interfaces (APIs), and are combined in an open architecture (OA) in which components with comparable interfaces can be substituted for each other [Ore00]. Under this strategy the development organization becomes an integrator of components largely produced elsewhere, interconnected to achieve the desired result.

The resulting OA systems can achieve reuse benefits such as reduced costs, increased reliability, and potentially increased agility in evolving to meet changing needs. But rather than a single proprietary license as when acquired from a proprietary vendor, or a single OSS license as in uniformly-licensed OSS projects, the resulting system typically has no recognized single software license. Instead it has, strictly speaking, a virtual license [AAS09] composed of each component's rights and obligations for that component under its governing license. The rights available for the system as a whole are the intersection of the rights sets for each component. In some cases the licenses may produce conflicting obligations and this intersection is empty, leaving a system that cannot legally be used, distributed, or modified. An emerging challenge is to realize the reuse benefits of HtL systems while managing virtual licenses to ensure that the desired system rights are available for an acceptable set of obligations.

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 [OSI15], 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.

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.

The basic relationship between software license rights and obligations can be summarized as follows: if you meet the specified obligations, then you get the specified rights. So, informally, for the academic licenses, if you retain the copyright notice, list of license conditions, and disclaimer, then you can use, modify, merge, sub-license, etc. For MPL, if you publish modified source code and sub-licensed derived works under MPL, then you get all the MPL rights. And so forth for other licenses. However, one thing we have learned from our efforts to carefully analyze and lay out the obligations and rights pertaining to each license is that license details are difficult to comprehend and track—it is easy to get confused or make mistakes. Some of the OSS licenses were written by developers, and often these turn out to be incomplete and legally ambiguous; others, usually more recent, were written by lawyers, and are more exact and complete but can be difficult for non-lawyers to grasp. The challenge is multiplied when dealing with configured system architectures that compose multiple components with heterogeneous licenses, so that the need for legal interpretations begins to seem inevitable [FKM08, Ros05]. Therefore, one of our goals is to make it possible to architect software systems of heterogeneously-licensed components without necessarily consulting legal counsel. Similarly, such a goal is best realized with automated support that can help architects understand design choices across components with different licenses, and that can provide support for testing build-time releases and run-time distributions to make sure they achieve the specified rights by satisfying the corresponding obligations.

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

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

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

The dag calculation algorithm follows the steps of legal reasoning (formalized to support automation) by which an informed analyst would reason out the results. Thus the traversals follow inference paths that follow (in more detail) the paths by which an analyst reasons out the same conclusions.

Structuring legal arguments

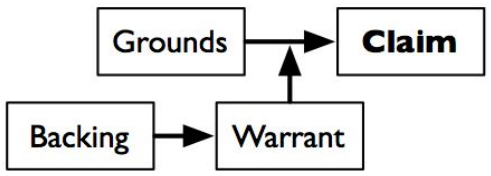


Figure 1. A claim, supported by grounds, their pertinence to the claim justified by a warrant, whose validity is supported by backing (diagram after [TRJ84])

The most influential approach for structuring legal arguments is that of Toulmin, who classified the parts of arguments into claims, grounds, warrants, backing, qualifiers, and rebuttals, in a recursive structure with a diagrammatic notation outlined in Figure 1 [TRJ84]. His approach has spread beyond the area of legal arguments and is used in general rhetoric and computer science. Toulmin divides arguments into

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

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

Licenses and Software Architectures

Open architecture (OA) software is a customization technique introduced by Oreizy [16] that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. Almost a decade later, we see more and more software-intensive systems developed using an OA strategy not only with open source software (OSS) components but also proprietary components with open APIs (e.g. [UEULA08]). Developing systems using the OA technique can lower development costs [ScA08]. Composing a system with HtL components, however, increases the likelihood of liabilities stemming from incompatible licenses. Thus, in this paper, we define an OA system as a software system consisting of components that are either open source or proprietary with open API, whose overall system rights at a minimum allow its use and redistribution.

OA may simply seem to mean software system architectures incorporating OSS components and open application program interfaces (APIs). But not all such architectures will produce an OA, since the available license rights of an OA depend on: (a) how and why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the licenses of different OSS components encumber all or part of a software system's architecture into which they are integrated [AIA08, ScA08].

Certain classes of architectural features affect the application and propagation of license provisions. These are identified by examining the software licenses of interest to see how their rights and obligations vary depending on the architectural context in which they are applied. A software architecture is composed of components, each of which is a “locus of computation and state” in a system, and connectors which link them and mediate interactions between them. In our research we have examined nearly 50 OSS and proprietary licenses and terms of service. We find that license analysis can be affected by:

- The type of connector used between two components, for example a static link, dynamic link, or client-server connection.
- The type of component, for example a source code component for which the sources are available so that the component can be rebuilt, corrected and evolved; a binary component for which source files are not available; a configured subsystem that interacts with the overall system as a single unit, but which may have its own architecture, components, and connectors; or a software service operating through an interface that hides what kind of component is providing the service.
- Whether a component is part of a separate library or is developed as an integral part of the system.
- The development history of evolved source code components; changes in which licenses they are released under, and evolutions in later versions of those licenses; and the specific copyright and license notices that are or were present at key points.
- Further distinctions made by other licenses not considered yet.

And so forth. The distinctions are limited only by the specifics of the licenses of interest, so in principle it is not possible to make an exhaustive list. However we have examined not only the most frequently used OSS licenses but also the most important classes of licenses, so we are confident in the general outlines and future validity of the automated analyses that we have developed.

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).

Heuristics for Designing HtL Systems

HtL system designers have developed 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 for specific licenses (depending on how the license provisions are interpreted). Rather than connecting conflicting components directly through static build-time links, the connection is made through a dynamic link, client-server protocol, license shim, or run-time plug-in.

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

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

License Rights and Obligations

A particularly knotty challenge is the problem of heterogeneous licenses in software systems. In order to illuminate the specifics of this challenge and provide a basis for addressing it, we analyzed a representative group of common OSS licenses and (for contrast) a proprietary license, using an approach based on Breaux's semantic parameterization [BAD08].

The stages of the analysis were:

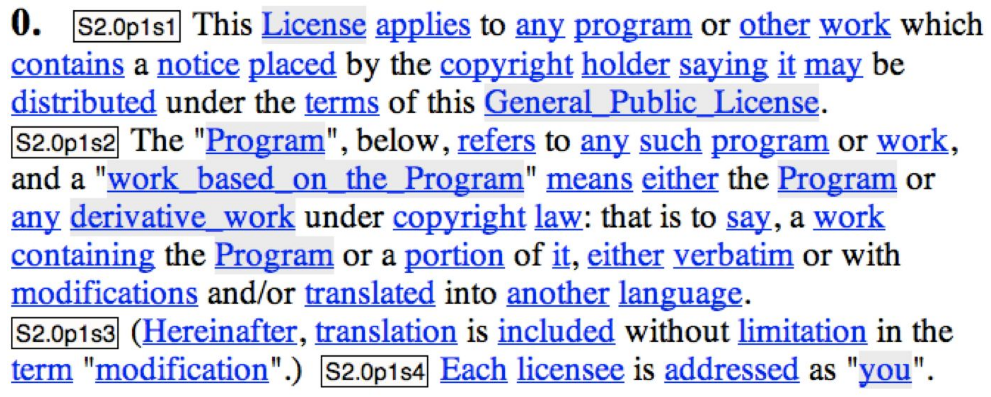

0. S2.0p1s1 This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General_Public_License.
S2.0p1s2 The "Program", below, refers to any such program or work, and a "work_based_on_the_Program" means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

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

1. First we disambiguated forward and backward references, identified synonyms, and distinguished polysemes that expressed different meanings with identical wording. We identified terms of art from copyright law, such as "Derived Work", and specialized terms defined for a particular license, such as "work based on the Program" for GPL and "Electronic Distribution Mechanism" for MPL. From this we constructed (automatically) a concordance to aid us in the remainder of the analysis. The concordance indexed the instances of each distinguished word term, excluding minor words such as articles, conjunctions, and prepositions whose use in a particular license carried no specialized meaning, and tagged each sentence with its section,

paragraph, and sentence sequence numbers. Figure 2 shows a portion of the concordance for GPL.
2. Next we identified the parts of each license that had no legal force, such as GPL 2’s “Preamble” section, or that dealt with any rights or obligations other than those for copyright, such as patents, trademarks, implied warranty, or liability, iterating with the concordance to confirm the identifications. The remainder of our analysis focused on copyright.
3. Using the concordances across the licenses, and guided by legal work on OSS licenses [Det06, FKM08, Ros05, StL04, Sto05], we identified words and phrases with the same intensional meaning, and textual structures parallel among the licenses. From these we iterated to identify natural language patterns each of which could be used as a restricted natural language statement (RNLS) to express the licenses.

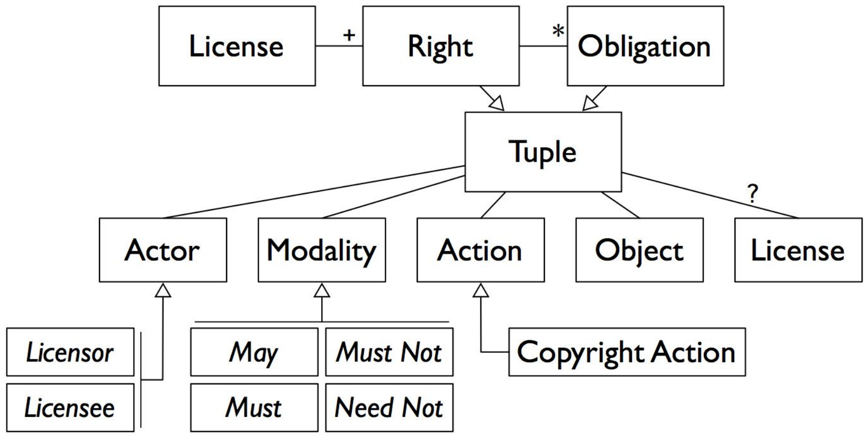


Figure 3. Metamodel for software licenses

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

We found a wide variety of license actions, some of which are defined in copyright law or derived from it and are distinguished as copyright actions..

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

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

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

Licensee · must · retain the [BSD] copyright notice in [file.c] where “retain the <License> copyright notice” is an action that is not a copyright action, BSD is the concrete license the action references, and file.c is the concrete source file the action references. The RNLS actions are defined with tokens identifying where the tuple’s objects are inserted, for example in the GPL action “sublicense OBJECT under LICENSE”. Figure 4 is an informal illustration of how actions may contain concrete objects and licenses, references to objects or licenses bound elsewhere, or quantifiers using the information in the license architecture abstraction described below to produce sets of rights or obligations.

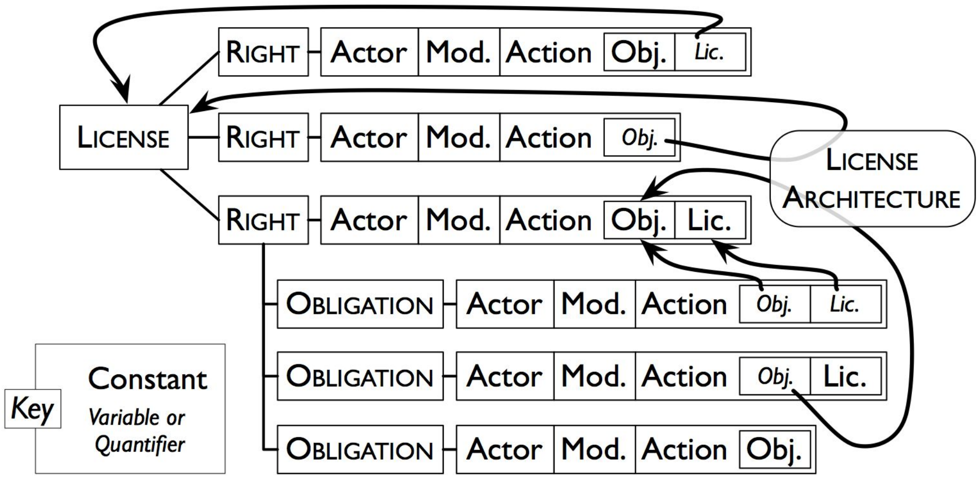
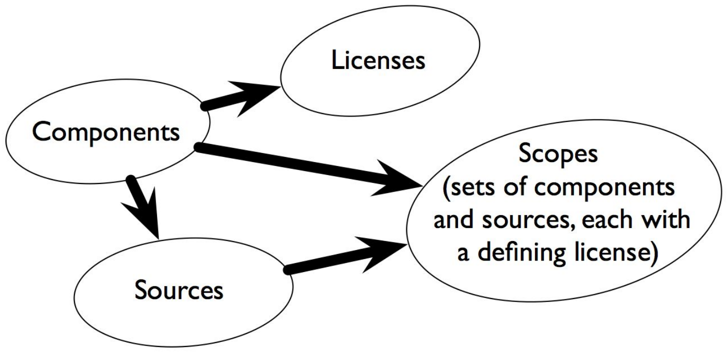


Figure 4. Object/license references, informally

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

1. the set of components of the system;
2. the relation mapping each component to its license;
3. the relation mapping each component to its set of sources; and
4. the relation from each component to the set of components in the same license scope, for each license for which “scope” is defined (e.g. GPL), and from each source to the set of sources of components in the scope of its component (Figure 5).



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

Figure 5. The license architecture metamodel

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

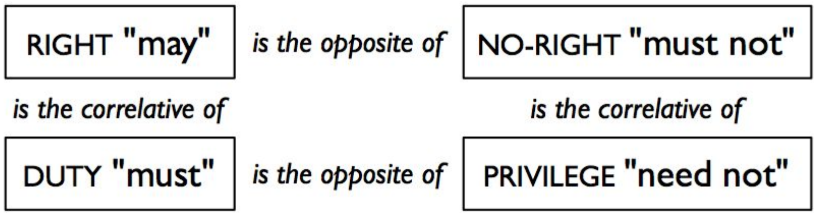


Figure 6. Hohfeld's four basic relations

We developed an approach for expressing software licenses that is more formal and less ambiguous than natural language, and that allows us to calculate rights and obligations for an HtL system and identify conflicts arising from the rights and obligations of two or more component's licenses. Our approach is based on Hohfeld's eight fundamental jural relations [Hoh13], of which we use right ("may"), duty ("must"), no-right ("must not"), and privilege ("need not") (Figure 6). Each relation has a correlative relation, which in our context relates an obligation to its necessary right:

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

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

<actor, modality, action, object>

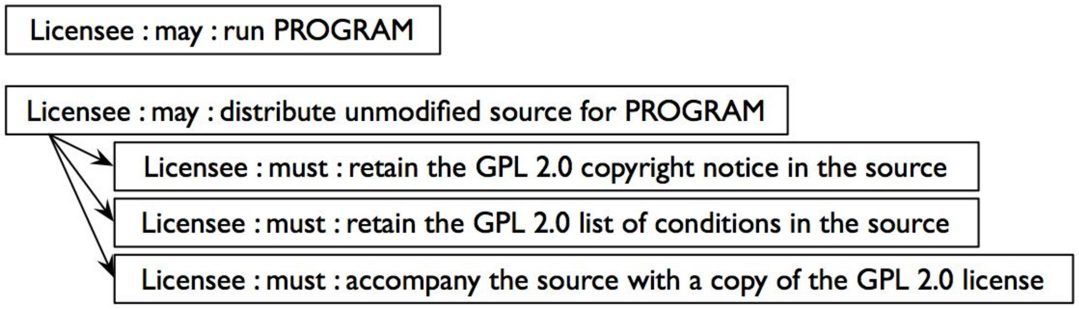

Licensee : may : run PROGRAM
Licensee : may : distribute unmodified source for PROGRAM
Licensee : must : retain the GPL 2.0 copyright notice in the source
Licensee : must : retain the GPL 2.0 list of conditions in the source
Licensee : must : accompany the source with a copy of the GPL 2.0 license

Figure 7. Some tuples for the GPLv2 license ***[UPDATE]***

The actor is either the "Licensee" or in a few cases "Licensor" for all the enactable, testable provisions of the licenses we have examined [ASA10]. The modality is "may" or "need not" for a right and "must" or "must not" for an obligation. The action is a verb phrase acting on zero or more objects, describing what may, need not, must, or must not be done. The objects are modules of the system or related artifacts such as a source file, the original version, documentation, and so forth. Typically a license right applies to any of a class of objects distributed under the license, such as any binary file or any modified source file; and the right's obligations will apply to the same object or a related object, such as the right's object's sources or the right's object's originals. For this reason we term rights and obligations as expressed in a license abstract, in contrast to a concrete right or obligation for one specific entity. Some actions are parameterized by a license as well.

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

A license is expressed as a set of rights, each right associated with zero or more obligations that must be fulfilled be granted it, and possibly a set of overall obligations that must be fulfilled for the license as a whole. Figure 7 sketches two rights from GPL version 2.0 (GPLv2), the first with no obligations and the second with three corresponding obligations.

The details of the license specification approach are described in our earlier work [AAS09, ASA10].

Applying Licenses to Software

Calculating the Inference DAG

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

1. r is not subsumed by any of the five copyright rights, and does not conflict with any general obligation of r's license L. In this case r is freely available.
2. r is subsumed by an abstract right R of the license, with e likewise subsumed by R's object. In this case all R's obligations O1 , O2 , . . . , On 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 8 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 O1 and O2 of R, which we apply to r's object e in order to obtain r's concrete obligations o1 and o2. Depending on what kind of object O1 has, o1 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 o1 we must have o1's correlative right r′1. The same considerations apply for O2, of course. The heavy arrow shows the flow of inference from desired concrete right through to required concrete obligations and correlative rights.

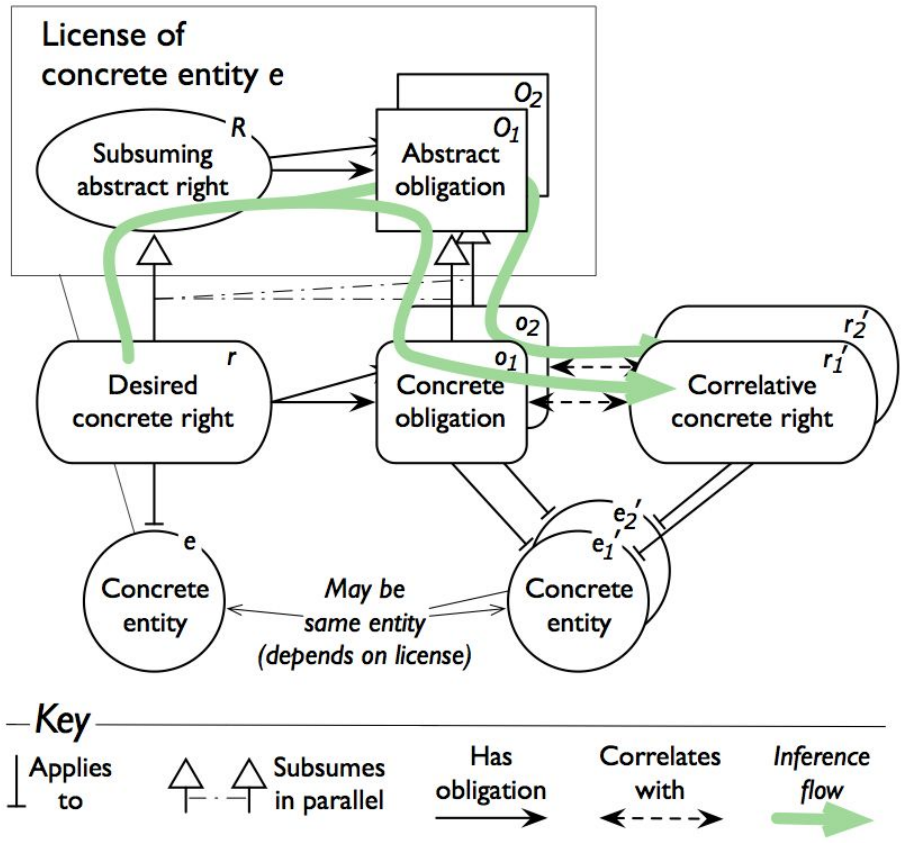


Figure 8. A step in a rights/obligations inference

If r′1 (r′2) is immediately available, its branch of the inference is complete. If not, the process recurses from r′1 (r′2).

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

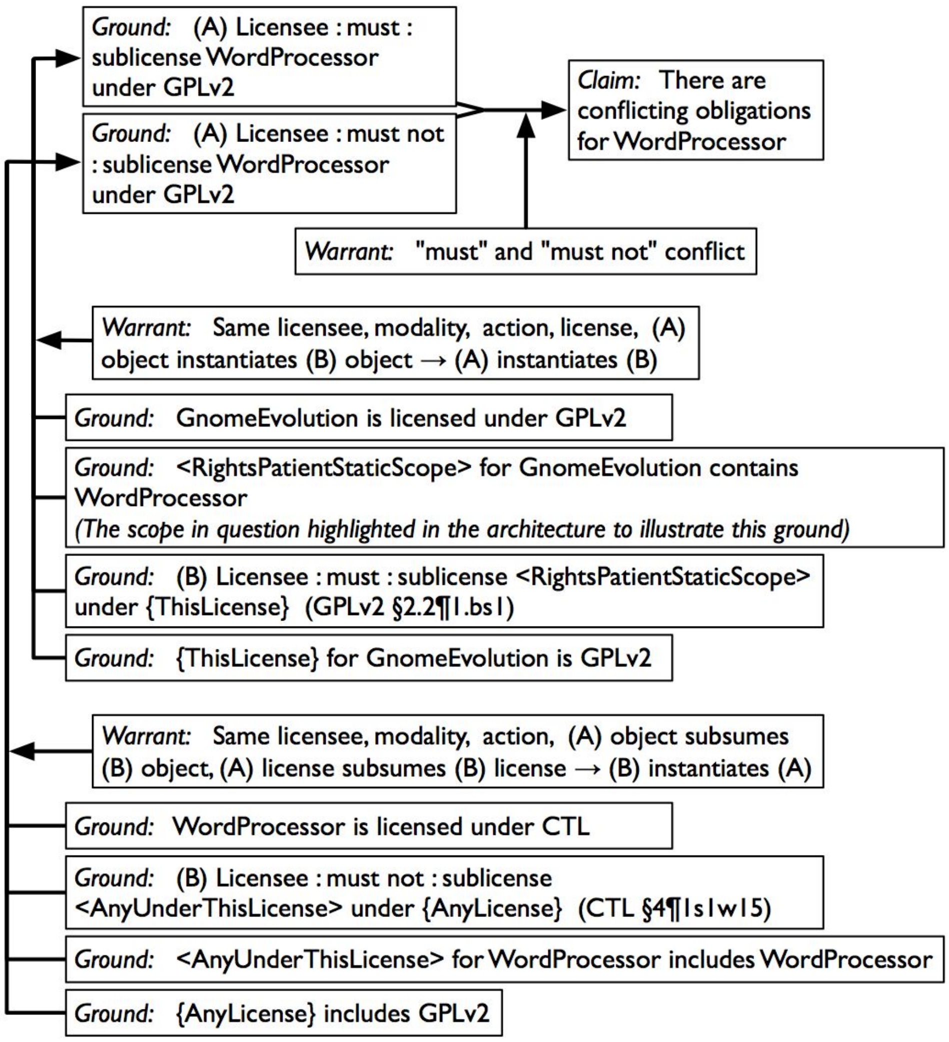

Ground: (A) Licensee : must : sublicense WordProcessor under GPLv2
Ground: (A) Licensee : must not : sublicense WordProcessor under GPLv2
Claim: There are conflicting obligations for WordProcessor
Warrant: "must" and "must not" conflict
Warrant: Same licensee, modality, action, license, (A) object instantiates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

Figure 9. Toulmin-structured arguments supporting (and explaining) a typical conflict between obligations for a GPLv2 and a proprietary component

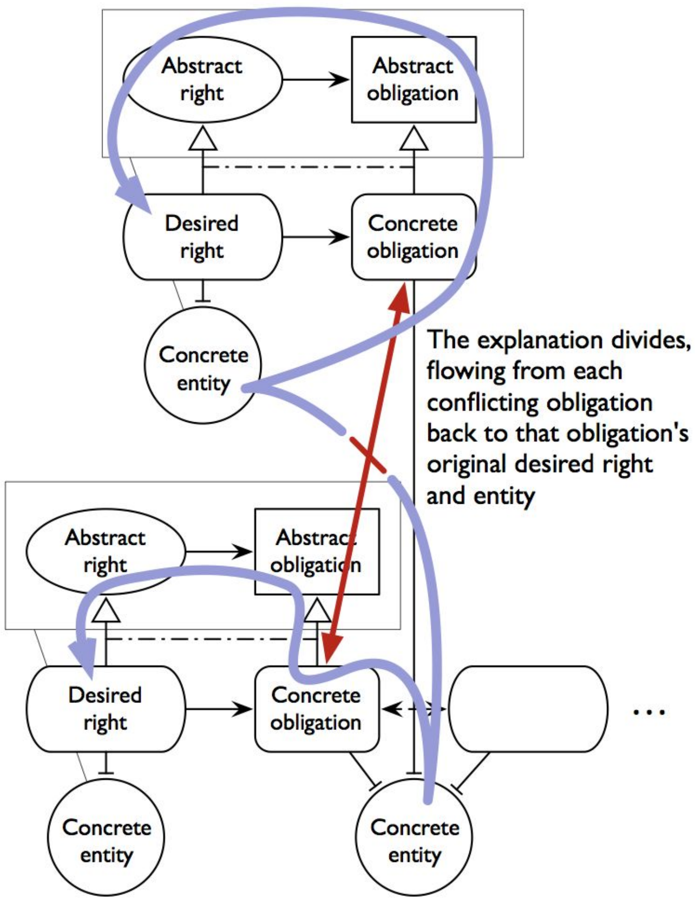
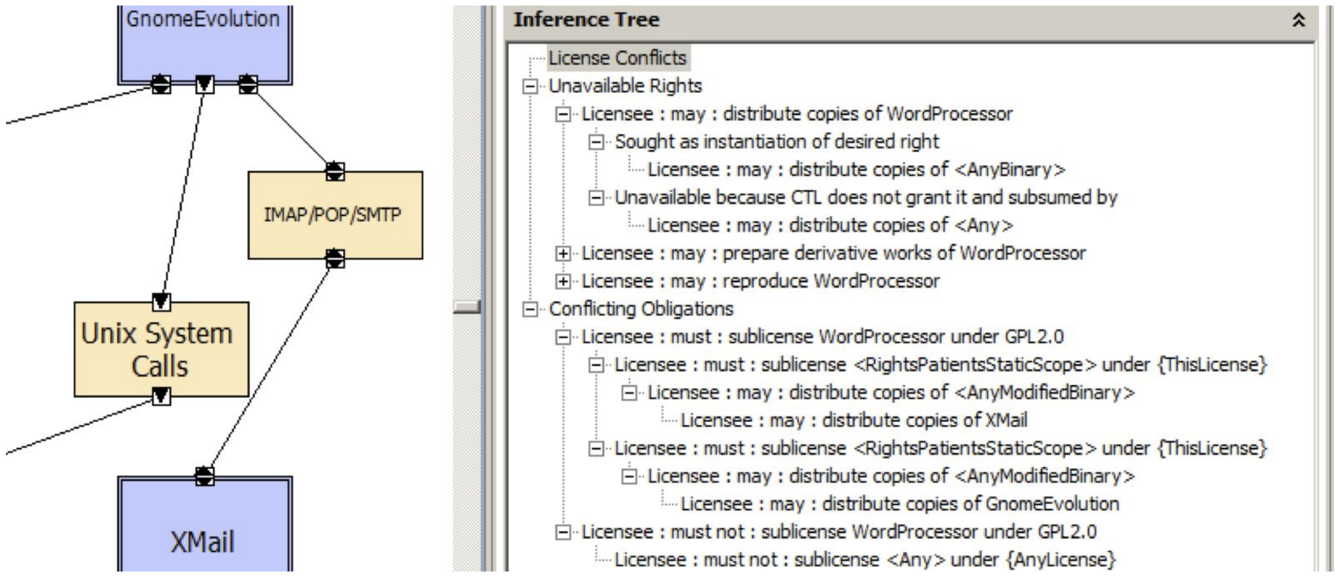


Figure SEKE11-7. Divided explanation flow for a conflict between two obligations

Figure 9 shows in Toulmin form a portion of an example inference that produces a conflict, involving a component e1 obtained under GPLv2 and modified, linked to a component e2 obtained under the proprietary Corel Transactional License (CTL) [Als15]. The architectural connection between e1 and e2 is one that is interpreted for this inference as propagating GPLv2 obligations, such as a static link. The right to distribute copies of the containing system is desired. In our prototype implementation (Figure 10) these arguments are presented in outline form, with the claim as the root of the outline and its grounds and warrant as its subheads, to be expanded as desired if further explanation is needed. A typical use would be:

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



Inference Tree

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

Figure 10. Prototype explanation results for a CTL-GPL2.0 conflict: (at top) unavailable rights (partially collapsed), (middle) two conflicting obligations.

Explanation by Argumentation

Figure SEKE11-7 shows the two explanation flows for a conflict between obligations. Each flow begins at the conflict and explains how one half of the conflicting pair came to be. The connection between the pair is straightforward, as they are identical except for their modalities which are always “must” for one and “must not” for the other.

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

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

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

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

Automation

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

The package is connected into the system design context by its integration into an ArchStudio 4 plugin [DAH07]. The plugin maps features of software architectures onto the license architecture abstraction needed for the virtual license calculation and displays results in the context of the architecture.

This approach provides:

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

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

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

The argument grounds drawn from the texts of licenses are implemented through URLs hyperlinking into our collection of software licenses tagged for reference with §-¶-sentence-word numbers [Als15]. Each URL cites the sentence or phrase from which a right or obligation arises. Word-level ids allow references to, for example, #S2.2p1.bs1w11 for the phrase beginning at word 11 of that sentence.

Discussion

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

A single license scheme

There is often a desire to specify a single license at architecture design time in order to insure a composed software system with single license compatible scheme at distribution time, and also at run time. Software licenses like GPL encourage this as part of their overall IP strategy for ensuring software freedom. Similarly, there is desire to determine whether a single known license can cover a designed or released system [GeH09]. However, a single license regime cannot in general be guaranteed to occur by chance; instead it is most effectively determined by design. In either case, it must be specified as a nonfunctional requirement for software development. But satisfying such a requirement limits the choice of software components that can be included in the system design and the system composition at distribution and run-time to those compatible (or subsumed) with the required overall system license. Consequently, our goal in this case is to insure a simple, homogeneous scheme relying on known licenses to determine the propagation and enforcement of their constraints.

A heterogeneous license scheme

In contrast to a single license scheme, a heterogeneous license scheme allows a software system to incorporate components with different IP licenses. Such a scheme gives more degrees of freedom than a single license scheme. For example, it allows for best-of-breed component selection, considering components with a range of licenses rather than only those with a specific license. It also allows for specification and design of software systems conforming to a reference architecture [BCK03]. This enables a higher degree of software reuse through inclusion of reusable software components that have a substantial prior investment in their development and use. Similarly, when relying on a reference architecture, design-time component choices need not be encumbered by license constraints, since the resulting system license rights and obligations need only be determined at distribution-time and runtime. Furthermore, the distribution and run-time system compositions are not limited to a single license; instead they are constrained only by the license rights and obligations that ensue for the entire system.

In a heterogeneous license scheme, the overall system rights and obligations can form a virtual license—a license whose rights and obligations can be determined, tested, and satisfied at any time, without being a previously approved license type, e.g. via the OSI license approval scheme [OSI15].

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

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

Conclusions

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

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

References

[Als15] Alspaugh, T.A *OSS (and other) licenses, §/¶/sentence/word-numbered.* http://www.thomasalspaugh.org/pub/osl-sps/ .

[AlA07] Alspaugh, T.A and Antón, A.I., (2007). Scenario Support for Effective Requirements, *Information and Software Technology*, 50(3), 198-220.

[AAS09] Alspaugh, T.A, Asuncion, H.U., and Scacchi, W. (2009). Intellectual property rights requirements for heterogeneously-licensed systems. In *17th Int. Requirements Engineering Conference* (RE'09), 24–33.

[ASA10] Alspaugh, T.A, Scacchi, W., and Asuncion, H. (2010). Software Licenses in Context: The Challenge of Heterogeneously Licensed Systems, *Journal of the Association for Information Systems*, 11(11), 730-755, November.

[BCK03] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, 2nd Edition, Addison-Wesley Professional, New York. 2003.

[BAD08] T. D. Breaux, T.D., Anton, A.I., and Doyle, J. (2008). Semantic parameterization: A process for modeling domain descriptions. *ACM Trans. on Softw. Eng. and Meth.*, 18(2), 1-44.

[DAH07] Eric Dashofy, E., Asuncion, H., Hendrickson, S., *et al.* (2007) Archstudio4: An architecture-based meta-modeling environment. In *28th Int. Conference on Software Engineering, Companion Volume*, 67–68.

[Det06] Determann, L. (2006). Dangerous liaisons—software combinations as derivative works? Distribution, installation, and execution of linked programs under copyright law, commercial licenses, and the GPL. *Berkeley Technology Law Journal*, 21(4).

[End13] Endres-Niggemeyer, B. (2013). The Mashup Ecosystem, in *Semantic Mashups: Intelligence Reuse of Web Resources*, Springer, 1-50.

[Fel07] Feldt, K. (2007). Programming Firefox: Building Rich Internet Applications with Xul. O'Reilly Media, Inc., 2007.

[FKM08] Fontana, R., Kuhn, B.M., Moglen, E., Norwood, M., Ravicher, D.B., Sandler, K., Vasile, J., and WIlliamson, (2008). A. *A Legal Issues Primer for Open Source and Free Software Projects*, Software Freedom Law Center, Version 1.5.1.
http://www.softwarefreedom.org/resources/2008/foss-primer.pdf

[GeH09] German, D. and Hassan, A.E. (2009). License integration patterns: Dealing with licenses mismatches in componentbased development. In *28th International Conference on Software Engineering* (ICSE '09), May 2009.

[Hoh13] Hohfeld, W.N. (1913). Some fundamental legal conceptions as applied in judicial reasoning. *Yale Law J.*, **23**(1):16–59.

[NeC06] Nelson, L. and Churchill, E.F. (2006). Repurposing: Techniques for reuse and integration of interactive systems. In *International Conference on Information Reuse and Integration* (IRI-08), 490-495.

[Ore00] Oreizy, P. (2000). *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution.* PhD thesis, Information and Computer Science Dept., University of California, Irvine.

[OSI15] OSI (2015). *Open Source Initiative*. http://www.opensource.org/.

[17] [Ros05] Rosen, L. (2005). *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall, Englewood Cliffs, NJ.

[ScA08] Scacchi, W. and Alspaugh, T.A. (2008). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *5th Annual Acquisition Research Symposium*, May 2008.

[StL04] St. Laurent, A. M. (2004). *Understanding Open Source and Free Software Licensing*. O'Reilly Media, Inc.

[Sto05] Stoltz, M.L. (2005). The penguin paradox: How the scope of derivative works in copyright affects the effectiveness of the GNU GPL. *Boston University Law Review*, 85(5):1439–1477.

[TRJ84] Toulmin, S., Rieke, R. and Janik, A. (1984). *An Introduction to Reasoning*. Macmillan. New York.

[UEU08] *Unity End User License Agreement*, Dec. 2008.
http:// unity3d.com/unity/unity-end-user-license-2. x.html

[USC08] U.S. Copyright Act, 17 U.S.C., 2008. http://www.copyright.gov/title17/

Chapter 6.

Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems

Chapter 6.

Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems[1]

Abstract
The role of software ecosystems in the development and evolution of open architecture systems whose components are subject to different licenses has received insufficient consideration. Such systems are composed of components potentially under two or more licenses, open source or proprietary or both, in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring. The software licenses of the components both facilitate and constrain the system's ecosystem and its evolution, and the licenses' rights and obligations are crucial in producing an acceptable system. Consequently, software component licenses and the architectural composition of a system help to better define the software ecosystem niche in which a given system lies. Understanding and describing software ecosystem niches for open architecture systems is a key contribution of this work. An example open architecture software system that articulates different niches is employed to this end. We examine how the architecture and software component licenses of a composed system at design time, build time, and run time help determine the system's software ecosystem niche and provide insight and guidance for identifying and selecting potential evolutionary paths of system, architecture, and niches.

Introduction

A substantial number of development organizations are adopting a strategy in which a software-intensive system (one in which software plays a crucial role) is developed with an open architecture (OA) [Ore00], 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 (Figure 1). With this approach, another organization often comes between software component producers and system consumers in order to compose and configure the produced components into a configured system. These organizations take on the role of system architect or integrator, either as independent software vendors, government contractors, system integration consultants, or in-house system integrators. In turn, such an integrator designs a system architecture that can be composed of components largely produced elsewhere, interconnected through interfaces accommodating use of dynamic links, intra- or inter-application scripts, communication protocols, software buses, databases/repositories, plug-ins, libraries or software shims as necessary to achieve the desired result.

An OA development process realizes or instantiates an ecosystem in which the integrator is influenced from one direction by the goals, interfaces, license choices, and release cycles of the

[1] An earlier version of this chapter appears in *Journal of Systems and Software*, 85 (2012). 1479-1494.

software component producers, and from another direction by the needs of the system's consumers. As a result the software components are reused more widely, and the composed OA systems can achieve reuse benefits such as reduced costs, increased reliability, and potentially increased agility in evolving to meet changing needs. However, an emerging challenge is to realize the benefits of this approach when the individual components are heterogeneously licensed [ASA10, GeH09, ScA08], each potentially with a different license, rather than a single OSS license as in uniformly licensed OSS projects or a single proprietary license as in proprietary development.

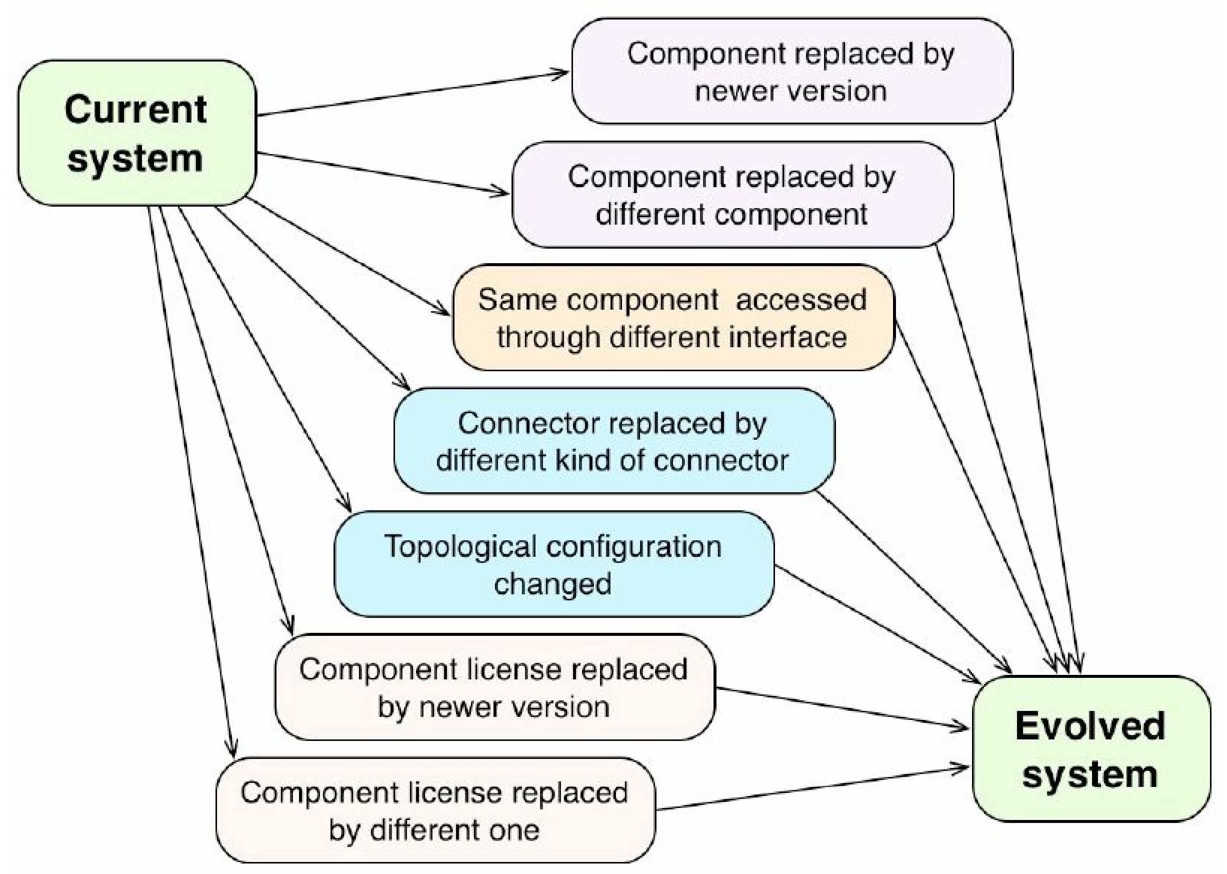

Current system
Component replaced by newer version
Component replaced by different component
Same component accessed through different interface
Connector replaced by different kind of connector
Topological configuration changed
Component license replaced by newer version
Component license replaced by different one
Evolved system

Figure 1. Paths of evolution for an OA system (described later).

This challenge is inevitably entwined with the software ecosystems that arise for OA systems (Figure 2). We find that an OA software ecosystem involves organizations and individuals producing, composing, and consuming components that articulate software supply networks from producers to consumers, but also:

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

These four items play a key role in defining the software ecosystem niche for a specific configured system—the specific software supply network that interconnects particular software producers of specific components, integrators, the software system architecture and its configured instantiation, and its consumers—as the remainder of this paper will make clear.

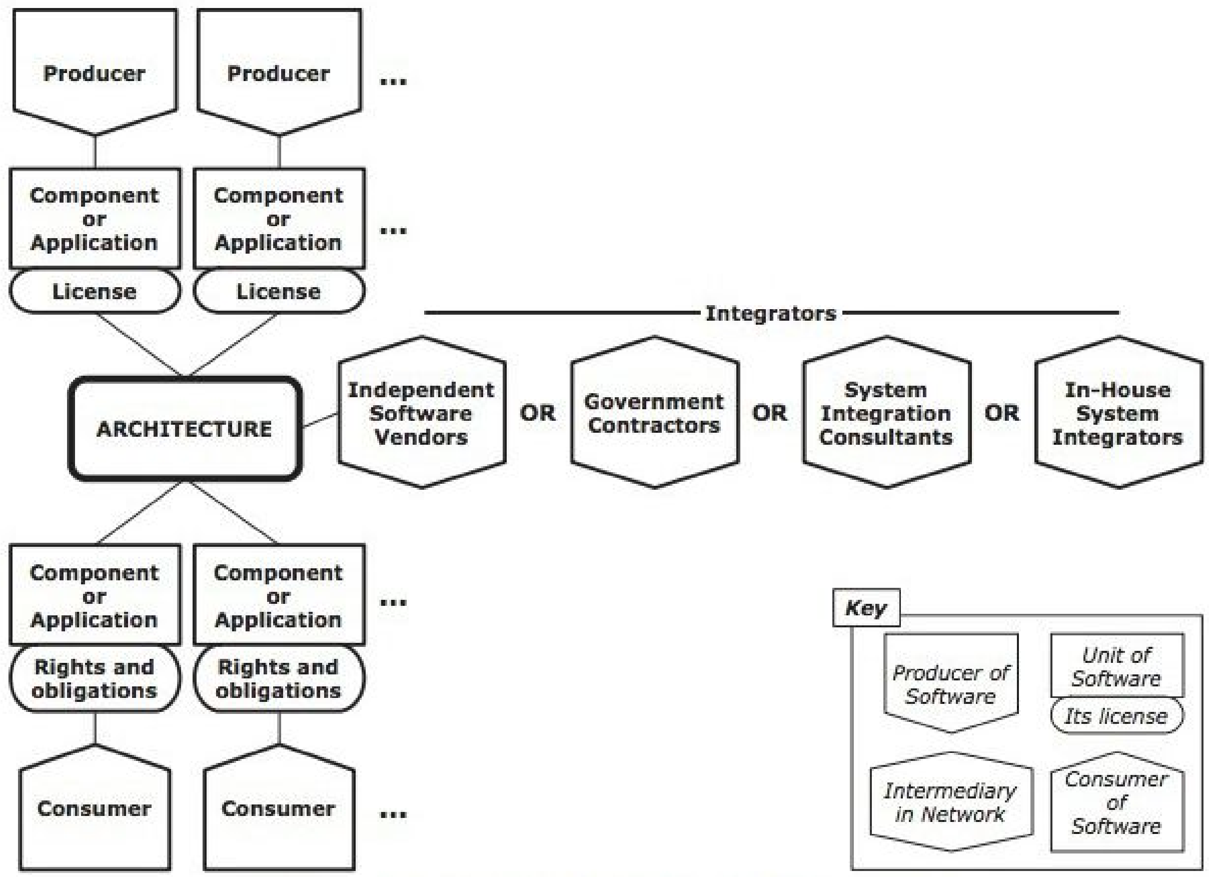


Figure 2. Schema for OA software supply networks (notation follows Boucharas, *et a*l., [BJB09]).

In our previous work [AAS09a, AAS09b, ASA10, ScA08], we examined how software licenses interact in significant ways through the software architecture of the system. Our approach, implemented in an Eclipse-based software architecture environment, automatically evaluates license conflicts in a software architecture and calculates the virtual license of rights and obligations for a composed system that result when its constituent components are licensed heterogeneously. With it architects directly examine the design decisions' licensing consequences: in the decision context, with enough information to identify definite license conflicts rather than only potential license conflicts, and early enough in the development process to make the right decision rather than correct a wrong one. This work contrasts with much practice and other research in which a configured system is examined after the fact, and often with substantial manual work by experts, to determine what licensing conflicts might exist in it. Here we build on our previous work by extending its context from software architecture to software ecosystems. The ecosystem context allows architects and integrators to examine potential evolution paths and the consequences of each one, with the ability to steer that evolution by specific changes to the system architecture and build- and run-time configuration.

The remainder of this paper is organized as follows. First, the next section motivates the work through a sequence of examples. This is followed by comparison and review of related research.

The next section discusses open architecture, followd by a discussion of the ecosystems that arise around open architecture systems. This gives rise to an examination of the evolution of software ecosystems, and then discussions of some implications that follow from this study. A summary of results then concludes the paper. Background on kinds of software licenses is presented in Chapter 4.

Motivating examples

Firefox: monolithically licensed

A few years ago, it was typical for a software system to be subject to a single intellectual property (IP) or copyright license covering the entire system, especially for proprietary software systems, but even for OSS. An example is the globally popular Firefox web browser, whose OSS is subject to the Mozilla Public License (MPL) version 1.1 [OSI11]. More recently, the Mozilla organization has updated its licensing strategy so that new OSS it produces is “tri-licensed.” This allows a licensee the choice to access, modify, and redistribute these systems under terms and conditions specified in either MPL, the GNU Project’s General Public License (GPL), or the Lesser General Public License (LGPL) [OSI11], while Firefox as a software product is under MPL. Users of Firefox and developers utilizing Firefox as a single component of larger systems need not concern themselves with whether the Mozilla organization has sufficient legal rights to all the Firefox code; Mozilla has assumed that responsibility.

Unity: heterogeneously licensed, closed architecture

These days, a growing segment of software systems are subject to multiple licenses, some of which may indicate potentially conflicting terms and conditions in different licenses, rather than to a single monolithic license. For example, the Unity game development tool, produced by Unity Technologies, is subject to multiple licenses [UnT08]. Its license agreement, from which we quote below, comprises a proprietary license for the core Unity software and presumably for the entire Unity system, plus at least 15 distinct licenses for at least 26 externally produced components, groups of components, and libraries, at least one of which has been further extended by Unity:

1. The Mono Class Library, Novell, Inc., MIT license,
2. The Mono Runtime Libraries, Novell, Inc., LGPLv2 (updated),
3. Boo, Rodrigo B. Oliveira, BSD license variant,
4. UnityScript, Rodrigo B. Oliveira, BSD license variant,
5. PhysX physics library, Novell Inc., proprietary,
6. libvorbis, Xiph.org Foundation, BSD license variant,
7. libtheora, Xiph.org Foundation, BSD license,
8. zlib general purpose compression library, Jean-loup Gailly and Mark Adler, inferred zlib/libpng license,
9. libpng PNG reference library, three individuals and Group 42 Inc., inferred zlib/libpng license,
10. jpeglib JPEG library, Thomas G. Lane, custom OSS license,
11. Twilight Prophecy SDK, Twilight 3D Finland Oy Ltd., inferred zlib/libpng license,

12. dynamic bitset, Chuck Allison and Jeremy Siek, custom OSS license,
13. The Mono C# Compiler and Tools, Novell, Inc., GPLv2 updated,
14. libcurl, Daniel Stenberg, MIT license derivative,
15. PostgreSQL Database Management System, University of California and PostgreSQL Development Group, BSD license derivative,
16. FreeType, The FreeType Project, FreeType License,
17. NVIDIA Cg compiler, NVIDIA Corp., GPLv2,
18. Scintilla and ScITE [source code editing), Neil Hodgson, Scintilla License,
19. 7-Zip Command [source code editing), Igor Pavlov, LGPLv2 [updated),
20. AES code [encryption/authentication), Brian Gladman, BSD license derivative,
21. FreeImage library, FreeImage project, FreeImage Public License,
22. Little CMS color management engine, Marti Maria Saguer, MIT license,
23. paintlib, Ulrich von Zadow and others, paintlib license,
24. Ericsson Texture Compression, Ericsson, proprietary license,
25. Particle Trimmer, Emil Persson, custom OSS license,
26. MonoDevelop IDE, MonoDevelop project and Unity, MIT license.

The overall software product license grants the right to install and use Unity but no rights to view or modify its source code [except for those components that are open source) or its design artifacts. Ordinarily the use of a properly licensed copy is unrestricted unless the software is patented; it is not clear whether any of Unity is patented or not, but as is often the case for proprietary licenses the Unity license states that unlicensed use is prohibited. Parts of the license explicitly give the user responsibility for obtaining any licenses required for (presumably future) patents that the software may infringe; trademarks are not mentioned except when reserving rights to them. Furthermore, an external developer or integrator has no access to Unity's architecture, and so cannot tell whether/how the separate license obligations for the externally produced components propagate to the obligations for Unity as a whole. However, the presence of a component with a reciprocal license that can propagate obligations to other components (17: NVIDIA Cg compiler, GPLv2) raises the necessity for Unity Technologies to have addressed these obligations architecturally in order for an end user not to propagate them further if using Unity as a component of a larger system.

The software ecosystem for Unity as a standalone software package is delimited by the diverse set of software components listed above (Figure 3). However the architecture that integrates and configures these components is closed: the architecture has not been made public, and much of the system is proprietary so that even what could be inferred from the source code cannot be determined. Thus consumers cannot determine the manner in which the different licenses associated with these components impose obligations or provide rights to consumers, or on the other components to which they are interconnected. Since there are several interpretations of some important OSS license provisions, this may be significant; did Unity Technologies firewall GPLv2's propagating obligations with dynamic links (following one well-supported interpretation of GPLv2) or more strongly with client-server connections (following another well-supported but more cautious interpretation)? A development organization with their own legal interpretation of GPLv2 and considering using Unity as a critical element of a composed system may need to know.

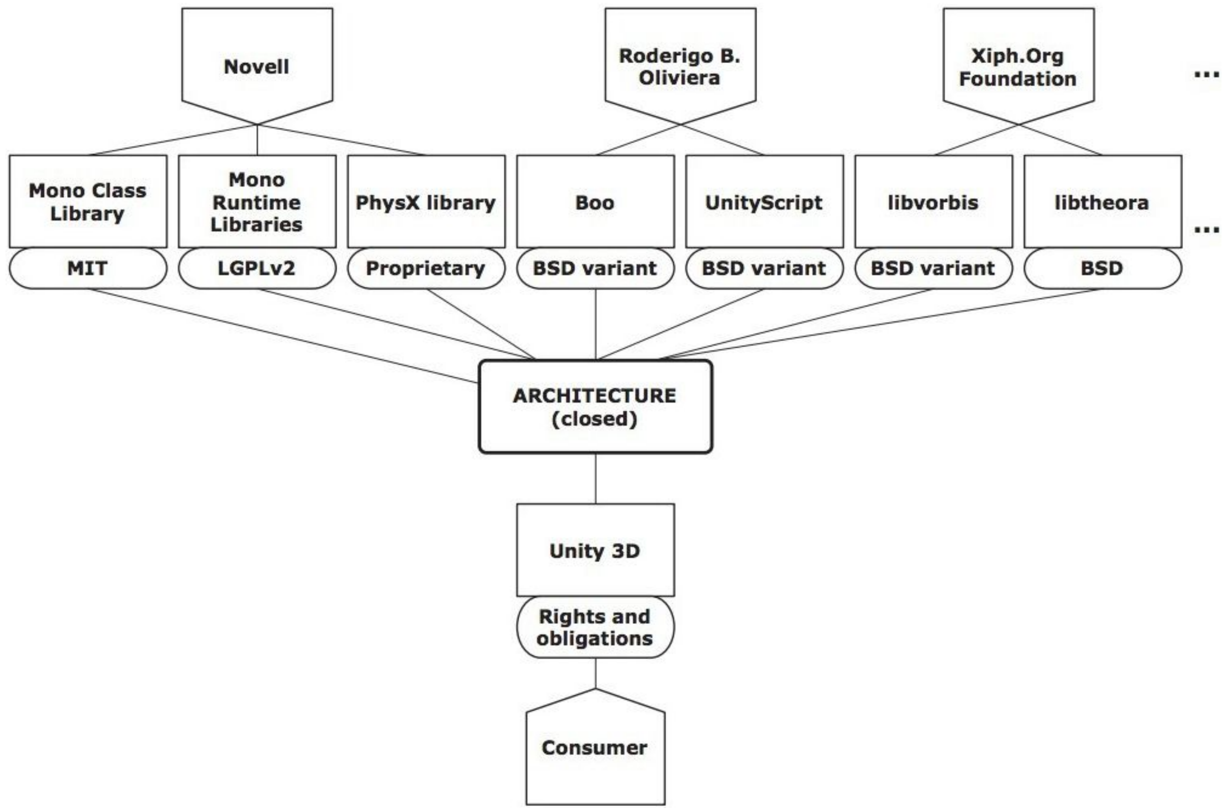


Figure 3. Ecosystem for Unity game development tool (partial).

As a consequence, there are several important questions that can't be answered about this ecosystem, but that an open architecture ecosystem annotated with software licenses and connector types can and should answer.

• What is Unity's *virtual license*, the set of rights available for the entire system and obligations demanded in exchange for those rights [AAS09b]?
• What portions of Unity do the various listed licenses pertain to, especially licenses such as the GNU General Public License that can propagate obligations along architectural connections to other components?
• What components of Unity can be evolved to later versions or replaced by similar components, in order to evolve the system toward more desirable functionality, desired software qualities, or more advantageous ecosystem and system evolution possibilities?
• For each component, how much of that component is being used by Unity? In other words, what interface is Unity using the component through? What other components support that interface, and what shims are available or could be developed to bridge the gap between that interface and the interfaces of other desirable components and versions?
• How and to what extent is Unity vulnerable to:
• potential litigation for license violations for copyright or copyleft infringements, or
• coercion due to dependence on specific development libraries and development or configuration tools?

There are also questions that cannot be answered even for an OA ecosystem, due to the differences between copyright law, under which an author gains specific exclusive rights for a specific term of years by the act of creation, and patent law, under which other inventors may unpredictably be granted new exclusive rights in the future over previously unencumbered parts of others' software systems, and also with trademark law, whose provisions are temporally dynamic and less uniform internationally.

- How and to what extent is Unity vulnerable to threats of patent infringement suits, whether for actual infringement, to force a settlement to avoid a lengthy, expensive, and risky court battle, or to persuade a system/platform vendor to engage in a crosslicensing agreement along with payment of license fees?
- How and to what extent is Unity vulnerable to co-opting of needed trademarks in some jurisdiction?

Google Chrome: heterogeneously licensed, open interfaces

The Google Chrome web browser represents yet another software ecosystem whose boundaries are defined in part through its use of externally licensed OSS components, that can be compared to Firefox and Unity. The license for Google's Chromium project [Chr11], from whose code base the Google Chrome browser is primarily built, comprises the BSD license for the Chromium core developed specifically for Google Chrome, plus 27 external components and libraries [some used only for specific platforms) under 14 distinct licenses:

1. bsdiff, BSD Protection License,
2. bspatch, BSD Protection License,
3. bzip2, BSD License,
4. dtoa, BSD License
5. ffmpeg, LGPL
6. HarfBuzz, MIT License,
7. hunspell, MPL 1.1 or GPL 2.0 or LGPL,
8. ICU, ICU License
9. JSCRE, BSD License,
10. libevent, BSD License,
11. libjpeg, libjpeg License,
12. libpng, libpng License,
13. libxml, MIT License,
14. libxslt, MIT License,
15. LZMA SDK, Special Exception License,
16. modp b64, BSD License,
17. Mozilla interface to Java Plugin APIs, MPL 1.1 or GPL 2.0 or LGPL,
18. npapi, MPL 1.1 or GPL 2.0 or LGPL,
19. nspr, MPL 1.1 or GPL 2.0 or LGPL,
20. nss, MPL 1.1 or GPL 2.0 or LGPL,
21. Pthreads for win32, LGPL 2.1,
22. Skia, Apache License 2.0,

23. sqlite, Public domain dedication,
24. V8 assembler, BSD License,
25. WebKit, BSD or LGPL 2 or LGPL 2.1,
26. WTL, Microsoft Public License [Ms-PL),
27. zlib, zlib License.

Two of the libraries (libpng and zlib) are also used by Unity though possibly under different licenses, and one component (LZMA) is part of a Unity component (7-Zip).

An examination of the component licenses shows that no Chromium component is subject to a proprietary license (MS-PL, despite its name, is a permissive open source license) and every one of the external Chromium components is available under a license that does not propagate license obligations to other components. Every component that is licensed under GPL, which can propagate obligations to other components depending on the connectors and architectural configuration around them, is also available under a non-propagating license such as MPL. It is evident that Google has chosen a policy of avoiding components licensed only under GPL and similar reciprocal licenses, forgoing the much broader selection of GPL-licensed components (approximately half of all open-source software is licensed under GPLv2) in exchange for not needing to consider architectural interactions among components, or whether any subsequent development or integration of Chromium can virally propagate GPLv2 obligations into other systems or applications.

It appears that all the external components have open interfaces (i.e. public and standardized), so that Chromium can evolve by replacing components with others implementing the same interfaces, or shimmed to them, as long as the replacements are also under non-propagating OSS licenses. However, Chromium's overall architectural composition, its architectural design, is (to our knowledge) not open and perhaps not even explicit.

IP License Considerations

Firefox, Unity, and Google Chrome have illustrated three related approaches to software licenses, software architecture, and software ecosystems.

Firefox is a monolithically licensed OSS system: all its code is given to the project under contributor license agreements [JeS11] that support releasing the entire project under a single license. External components are kept at arm's length, architecturally speaking, as plug-ins subject to their own licenses, with no license interaction with Firefox itself.

Unity is a closed system with externally produced components, some with open interfaces and OSS licenses and others with proprietary licenses. The external components retain their own licenses which are incorporated into the overall license for Unity either by reference or by quoting the license text. Unity Technologies has likely followed an internal, manual process for resolving potential license conflicts among components, so that it can offer Unity to its consumers without causing the suppliers of those components to object. Because Unity Technologies does not release Unity overall as an OSS project, most of the sub-licensing provisions of the components' licenses do not come into play, simplifying Unity Technologies' manual analysis for license conflicts at the expense of preventing licensees from modifying Unity to meet their own needs more exactly. Some of the components are OSS, and for one of them (26: MonoDevelop IDE) Unity Technologies' modifications are open as well, but Unity users cannot modify components themselves and rebuild a more capable version of Unity from them.

Google Chrome is an OSS system incorporating externally produced components. However, it is not an OA system since it does not appear to have a formally specified open architecture that explicitly composes components interlinked through connectors to derive or realize a buildable system configuration. Instead, as in most OSS projects, its parent project Chromium relies on an implicit architecture that cannot be completely identified and may only be assessed by reading the source code and reviewing online artifacts and developer interaction records (e.g. postings to a bulletin board, reviewing bug reports, checking comments in source compilation build scripts, or developer chat channels).

Because its architecture is implicit, the overall system license for Chromium cannot be calculated automatically [AAS09b, AAS11] but is instead compiled manually. The several years-old date of 2008 for the Chromium license, and the project's discussion of the change from JSCRE to the V8 regular expression engine [Chr09], for which the license was not updated, support this inference.

In order to simplify the process for resolving potential license conflicts, the Chromium project appears to have limited its external components to those available under non-propagating, non-reciprocal OSS licenses. The Chromium source is publicly available for perusal and modification, but not under a single monolithic license; each component is licensed under its own. Users can modify and rebuild Chromium to suit their own needs, as long as they meet the (separate) license obligations of all the components, and do not contravene Google trademark or branding restrictions.

In summary, none of these widely used systems provide enough information to completely evaluate potential evolution paths, or to automatically calculate overall IP rights and obligations. But this information and IP stipulations are needed to fully articulate the software supply networks that reveal which software ecosystem instances (or niches) each system exists within. In order to explore the issues raised by open architecture software ecosystems, it is necessary to consider a system about which the necessary information is available. We do not claim that only open architecture systems are important or useful, but rather that only such systems can take full advantage of the evolutionary and analytical opportunities OAs support. Because it is not possible, in general, to infer a system's software architecture after the fact, or to satisfactorily impose an OA on a system developed without one, the system must be designed from the beginning with an explicit architecture as a first-class development architecture. Consequently, we present an example system that utilizes a simple, archectypal open architecture below, in a later section. This system illustrates the issues that arise with more complex systems like Unity and Google Chrome, as well as additional possibilities not available without an OA, and does so with greater brevity and clarity.

Subsequently, we see that software ecosystems can be understood in part by examining relationships between architectural composition of software components that are subject to different licenses, and this necessitates access to the system's architecture composition. By examining the open architecture of a specific composed software system, it becomes possible to explicitly identify the software ecosystem niche in which the system is embedded.

Related research

Software ecosystems

The study of software ecosystems is emerging as an exciting new area of systematic investigation and conceptual development within software engineering. Understanding the many possible roles that software ecosystems can play in shaping software engineering practice is gaining more attention since the concept first appeared [MeS03]. Bosch [2009] builds a conceptual lineage from software product line (SPL) concepts and practices [Bos00, ClN01] to software ecosystems. SPLs focus on the centralized development of families of related systems from reusable components hosted on a common platform with an intra-organizational base, with the resulting systems either intended for in-house use or commercial deployments. Software ecosystems then are seen to extend this practice to systems hosted on an inter-organizational base, which may resemble development approaches conceived for virtual enterprises for software development [NoS99]. Producers of commercial software applications or packages thus need to adapt their development strategy and business model to one focused on coordinating and guiding decentralized software development of its products and enhancements (e.g. plug-in components, apps, mashups).

Relations among and within software ecosystems

Jansen *et al*. [JFN09a, JFN09bb] observe 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 [Sca07] for example, identifies that the world of OSS development is a loosely coupled collection of software ecosystems 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 [JeS05].

Boucharas *et al*. [BJB09] then draw 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 [Kue08] examines how Microsoft's software ecosystem developed around 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.

OSS ecosystems also exhibit strong relationships between the ongoing evolution of OSS systems and their developer and user communities, such that the success of one co-depends on the success of the other [Sca07]. Ven and Mannaert discuss the challenges independent software vendors face in combining OSS and proprietary components, with emphasis on how OSS components evolve and are maintained in this context [VeM08].

Next, 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”. 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 servicespecific 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.

Last, the MITRE Corporation and others in the Defense community seek to embrace the development of agile C2 systems [RBC12]. Such systems are envisioned to arise from the assembly and integration of system elements (application components, widgets, content servers, networking elements, etc.) within a software ecosystem of multiple producers, integrators, and consumers who may supply or share the results of their efforts. The assembly and integration of system elements produces “assembled capabilities” (AC) for C2 systems. AC may be produced, acquired, integrated, shared, or reused by different trusted parties. AC may address a set of ISR data/signal processing components, office productivity components supporting mission planning, or the like. Our purpose is to identify how our approach to the design of secure OA systems can be aligned with their vision for agile C2 systems. Along the way we focus on design of OA system capability involving office productivity components that must be configured as a secure AC.

The design and development of agile C2 systems follows from two sets of principals: one set addressing guidelines/tenets for multi-party engineering (MPE) of C2 system components; the other set addressing attributes of agile and adaptive ecosystems (AAE) for producing AC or C2 system elements. For brevity, we simply identify these principals for MPE and AAE, as they are more fully explained elsewhere [RBC12], but we do so in ways that foreshadow and more clearly align with our approach that follows in a later section.

MPE Tenets:

1. Provide small system components that can be rapidly developed, and accommodate different functionally equivalent variants, or functionally similar versions.
2. Certify components are consistent with “shared agreements” regarding security requirements, system architecture, data semantics, production and integration processes or process constraints, and other aspects of mission-specific or mission-common domain models.
3. Supply diverse C2 system components via a market of component producers or integrators.
4. Assemble and integrate AC from components available in the market that are consistent with relevant shared agreements.

5. Provide feedback from C2 system users to component producers or capability integrators to improve market efficiency and effectiveness.

AAE Attributes:
1. Encourage and sustain a software ecosystem that is agile (supports assembly and integration C2SC) from components in market, and adaptive (supports substitution of functionally similar 4 component versions or functionally equivalent component variants), in line with user feedback.
2. Component markets are federated so as to accommodate sharing, reuse, or trading of components across different system integrators or consumer organizations.
3. Shared agreements serve as a basis for enabling multi-party collaboration in system development, integration, and evolution/sustainability.
4. Production, integration, or post-deployment support for components or AC must be viable for small businesses or large, as well as promoting market diversity and effectiveness.
5. Consumer/user organizations seek to manage portfolios of components or AC that collectively improve mission effectiveness, agility and adaptiveness, while reducing costs.

Finally, to help understand what we mean by a software ecosystem, we refer to Figure 2 to represent where different parties are located across a generic software ecosystem, and the supply networks or multi-party relationships that emerge to enable the software producers to develop and release products that are assembled and integrated by system integrators for delivery to consumer/enduser organizations.

Software ecosystems and software product lines
Along with other colleagues [BBS10; BrB02, vPB10], Bosch also identifies alternative ways to connect reusable software components through integration and tight coupling found in SPLs, or via loose coupling using glue code, scripting or other late binding composition schemes in ecosystems or other decentralized enterprises [NolS99, NoS01], as a key facet that can enable software producers to build systems from diverse sources.

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 [SWZ12]: For example, we can find *functionally similar* alternatives from software (component) producers of web browsers like Mozilla (Firefox, Camino, Sea Monkey) vs. Google (Chrome) vs. Microsoft (Internet Explorer), vs. others. Similarly, for word processors, we find alternatives including Microsoft (Word) vs. abisoft.com (AbiWord) vs. Google (Google Docs, which is a remote Web service rather than a component), vs. 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 to modify, for example, OA system topology, 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 [NaS87, Bos06] or a product line [CN01].

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 [Fra10, SJW11]. Last, software product lines can be bound to a network of software producers, system integrators, and system users/consumers through a software ecosystem [Bos09], such that secure systems can be realized through composition or configuration at the software ecosystem level, as described in this chapter. 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 to address the challenges of creating secure OA systems through secured software product lines.

Building on related work
Our work in this area builds on these efforts in the following ways. First, we share the view of a need for examining software ecosystems, but we start from software system architectures that can be formally modeled and analyzed with automated tool support [Bos00, TMD09]. Explicit modeling of software architectures enables the ability to view and analyze them at design time, build time, or deployment/run time. Software architectures also serve as a mechanism for coordinating

decentralized software development across multi-site projects [ORM03]. Similarly, explicit models allow for the specification of system architectures using either proprietary software components with open APIs, OSS components, or combinations thereof, thereby realizing OA systems [ScA08]. We then find value in attributing open architecture components with their IP licenses [AAS09b], since software licenses are an expression of contractual/social obligations that software consumers must fulfill in order to realize the rights to use the software in specified allowable manners, as determined by the software's producers.

Open architecture

OA is a software design customization technique introduced by Oreizy [Ore00] that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. The technique was introduced and desribed in Chapter 2.

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.[UnT08]). Using this approach can lower development costs and increase reliability and function [ScA08]. 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 explicitly interconnected components that are either open source or proprietary with open APIs, whose overall system rights at a minimum allow its use and redistribution, in full or in part*.

It may appear that using a system architecture that incorporate OSS components and uses open APIs will result in an OA system. But not all such architectures will produce an OA, since the (possibly empty) set of available license rights for an OA system depends on: (a) how and why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the licenses of different OSS components encumber all or part of a software system's architecture into which they are integrated [ALA08, ScA08]. Thus, as noted earlier, neither Firefox, Unity, nor Google Chrome are OA systems, even though all three are built with OSS components. But how can we specify and design a system so that it does have an OA?

Each component selection implies acceptance of the license obligations and rights that the producer seeks to transmits to the components consumers. However in an OA design development, component interconnections may be used to intentionally (or unintentionally) propagate these obligations onto other components whose licenses may conflict with them or fail to match [AAS09b, GeH09]; the system integrator can decide to insert software shims using scripts, dynamic links to remote services, data communication protocols, or libraries to mitigate or firewall the extent to which a component's license obligations propagate. This style of build-time composition can be used to accommodate a system's consumers' choice to select components that either ensure or avoid certain licenses (for example Firefox's policy of only accepting source code that can be tri-licensed, or Google Chromium's apparent policy of excluding components governed by proprietary or strong-copyleft licenses, both of which were shown earlier), or that isolate the license obligations of certain desirable components. It also allows system integrators and consumers to follow a “best of breed” policy in the selection of system components. Finally, if no license conflicts exist in the system, or if the integrator and system consumer are satisfied with the component and license choices made, then the compositional bindings may simply be set in the most efficient way available. This realizes a policy for accepting only components and licenses whose obligations and rights are acceptable to the system consumers.

Understanding open architecture software ecosystems

A software ecosystem constitutes a software supply network that connects software producers to integrators to consumers, through licensed components and composed systems. Figure 4 illustrates a software ecosystem for an OA example system discussed below. By analogy to Hutchinson’s definition of a niche in a biological ecosystems as “an n-dimensional hypervolume ... every point in which corresponds to a state of the environment which would permit the species ... to exist indefinitely” [Hut57], we define software ecosystem niches below.

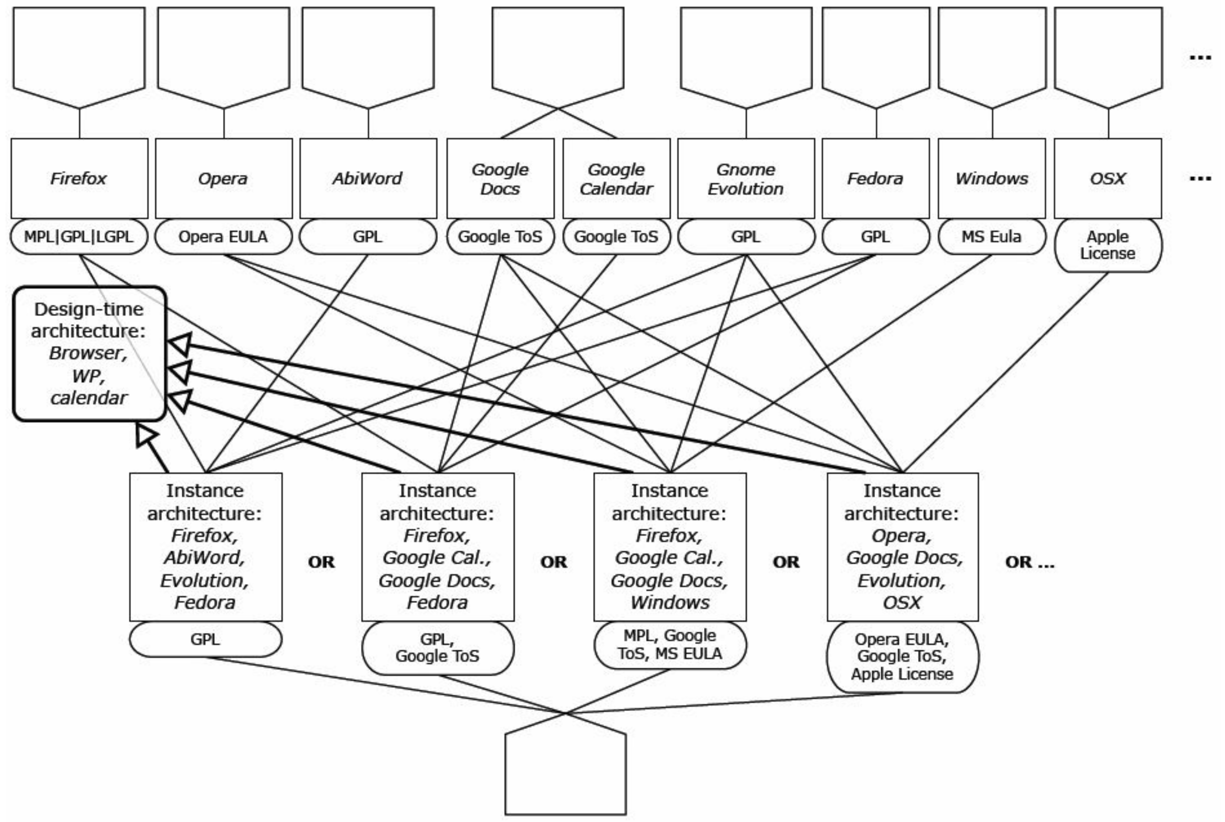

...
Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

Figure 4. Ecosystem for four possible instantiations of a single design architecture.

Software ecosystem niches

A software ecosystem niche articulates a specific software supply network that interconnects particular software producers of specific components, integrators, and consumers. The niche defined by a software system may lie within an existing single ecosystem, or it may span a network of several software producer ecosystems.

Firstly, a composed software system architecture largely determines the system’s software ecosystem niche, since the architecture identifies the components, their licenses and producers, and thus the network of software ecosystems in which it participates. Such a niche also transmits license-borne obligations and access and usage rights passed from the participating software component producers, through integrators, on to system consumers. Thus, system architects or component integrators help determine in which software ecosystem niche a given instance architecture for the system participates.

Secondly, system integrators can update or modify system architectural choices not only at design time, but also at build time, when components are joined together into an executable, or at run-time, when bindings to remote executable services are instantiated, thus shifting the instantiated system to a related but distinct niche.

As a software system evolves over time, as its components are updated or changed or their architectural interconnections are refactored, it is desirable to determine whether and how the system's ecosystem niche may have changed and how it can be advantageously steered for the future. Such a change implies at minimum that the software supply network may have been reconfigured, and thus obligations and rights passed from producers and integrators to system consumers may have also changed in some way. A system may evolve because its consumers want to migrate to alternatives from different component producers, or choose components whose licenses are now more desirable. Software system consumers may want to direct their system integrators to compose the system's architecture so as to move into or away from certain niches. Thus, understanding how software ecosystem niches emerge is a useful concept that links software engineering concerns for software architecture, system integration/composition, and software evolution to organizational and supply network relationships between software component producers, integrators and system consumers. It also helps articulate how the obligations and rights provided by producers are propagated/constrained by integrators onto system consumers as the system is developed and evolves.

An example system
To help explain how OA systems articulate software ecosystem niches, we provide a software architecture example system for use in this paper. This OA system utilizes a simple architectural design that composes a web browser, word processor, calendaring, and email applications, onto a host platform operating system, possibly with remote services for some components, designed and integrated by some organization and distributed to its consumers, some of whom may in turn integrate it into a larger system The same issues arise as if it utilized a graphics library, encryption module, typesetting engine, and thread management component instead, or with 400 components rather than 4, but this architecture illustrates the issues more simply and has the advantage of applying to many existing systems, including systems built by the authors.With these architectural elements, we can create an design-time or reference architecture for a system that conforms to the software supply network shown in Figure 4. This design-time architecture appears in Figure 5; note that it only specifies components by type rather than by producer, meaning the choice of producer component remains unbound at this point.

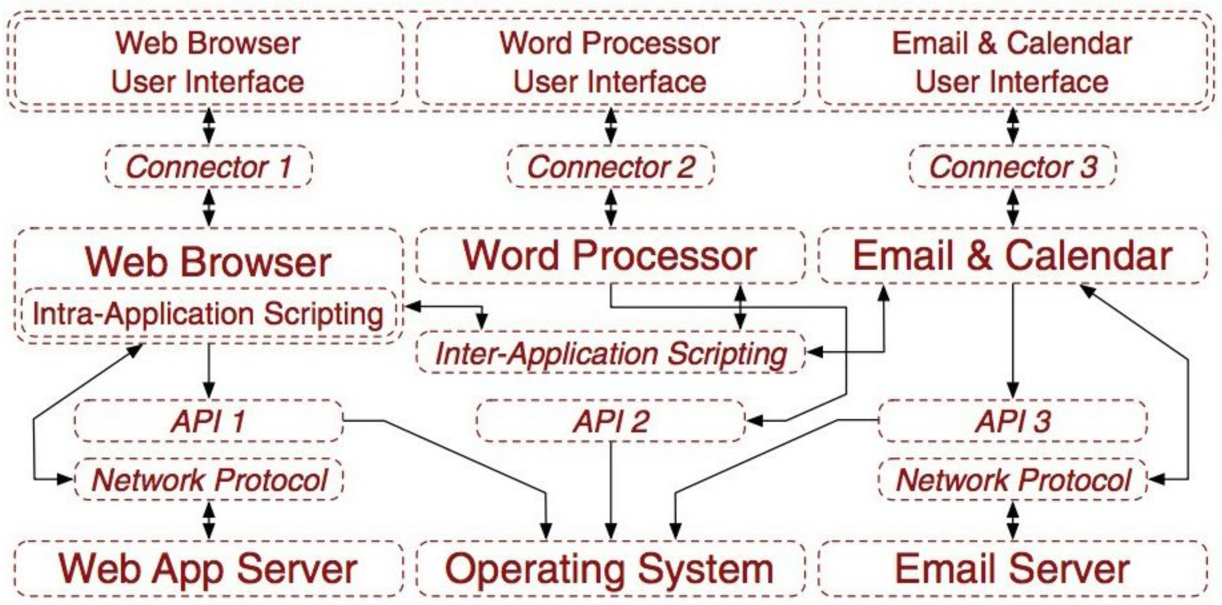


Figure 5. A design-time architecture.

Then in Figure 6, we create a build-time rendering of this architectural design by selecting specific components from designated software producers. The gray boxes correspond to components and connectors not visible in the run-time instantiation of the system in Figure 7.

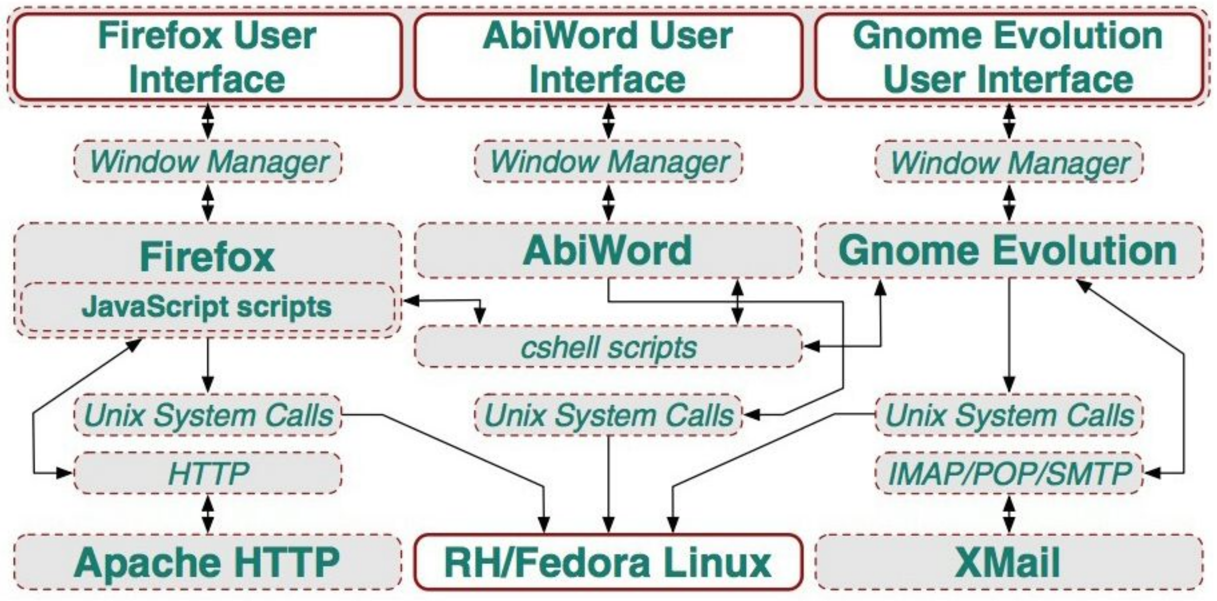
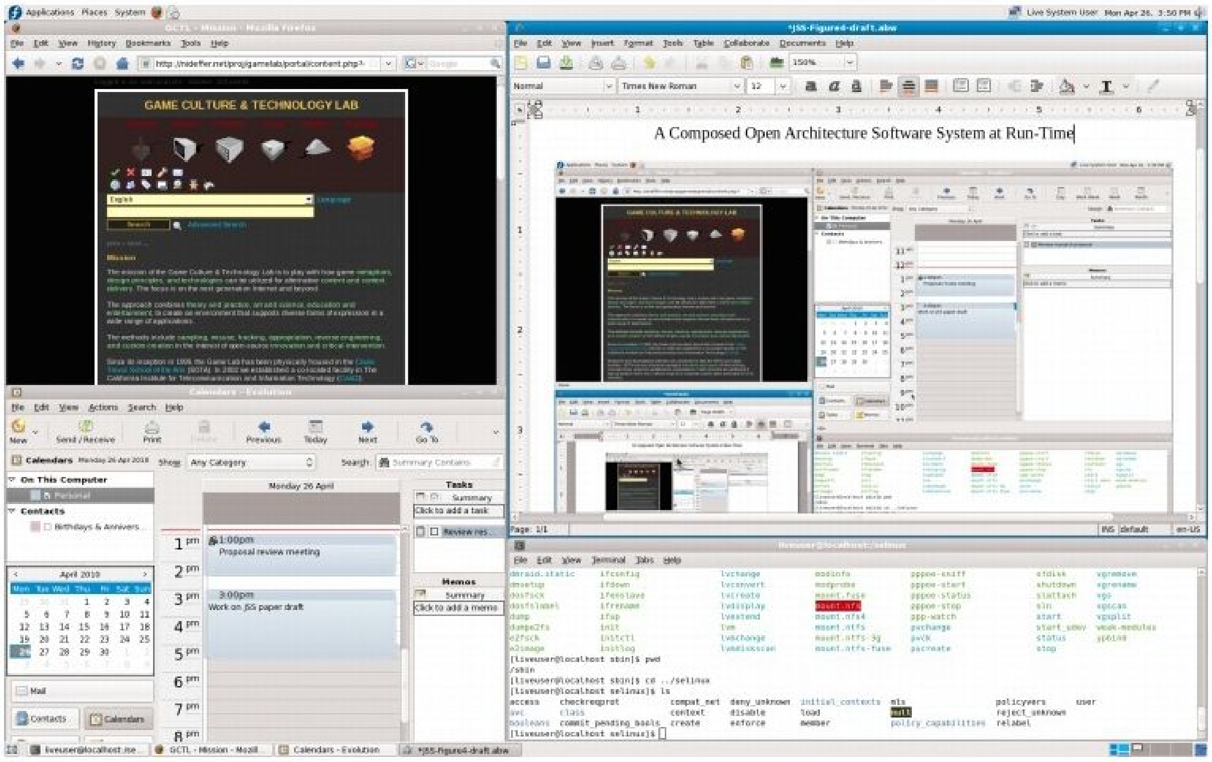


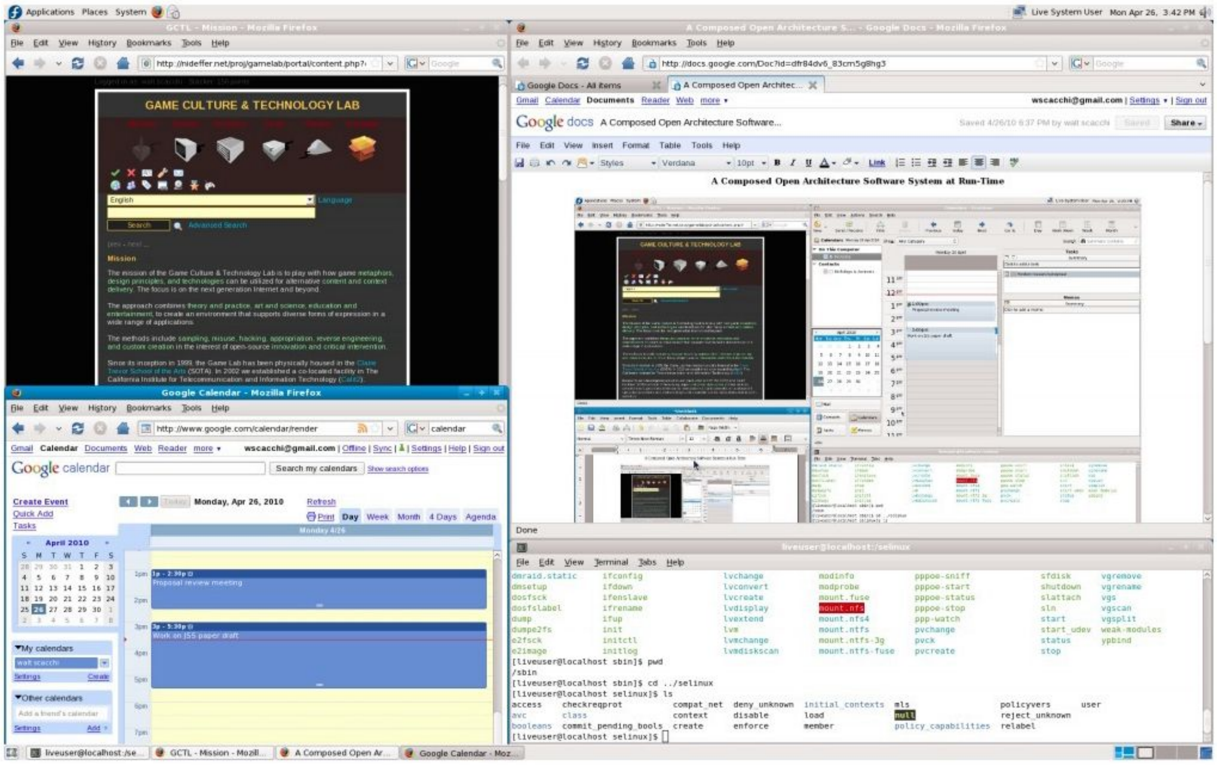
Figure 6. A build-time architecture.

Figures 7–9 display alternative run-time instantiations of the design-time architecture of Figure 5. The architectural run-time instance in Figure 7 corresponds to the software ecosystem niche shown in Figure 10; Figure 8 corresponds to the niche in Figure 11; and Figure 9 designates yet another niche different from the previous two. The run-time instantiations are then distributed to the consumers of the system.


A Composed Open Architecture Software System at Run-Time

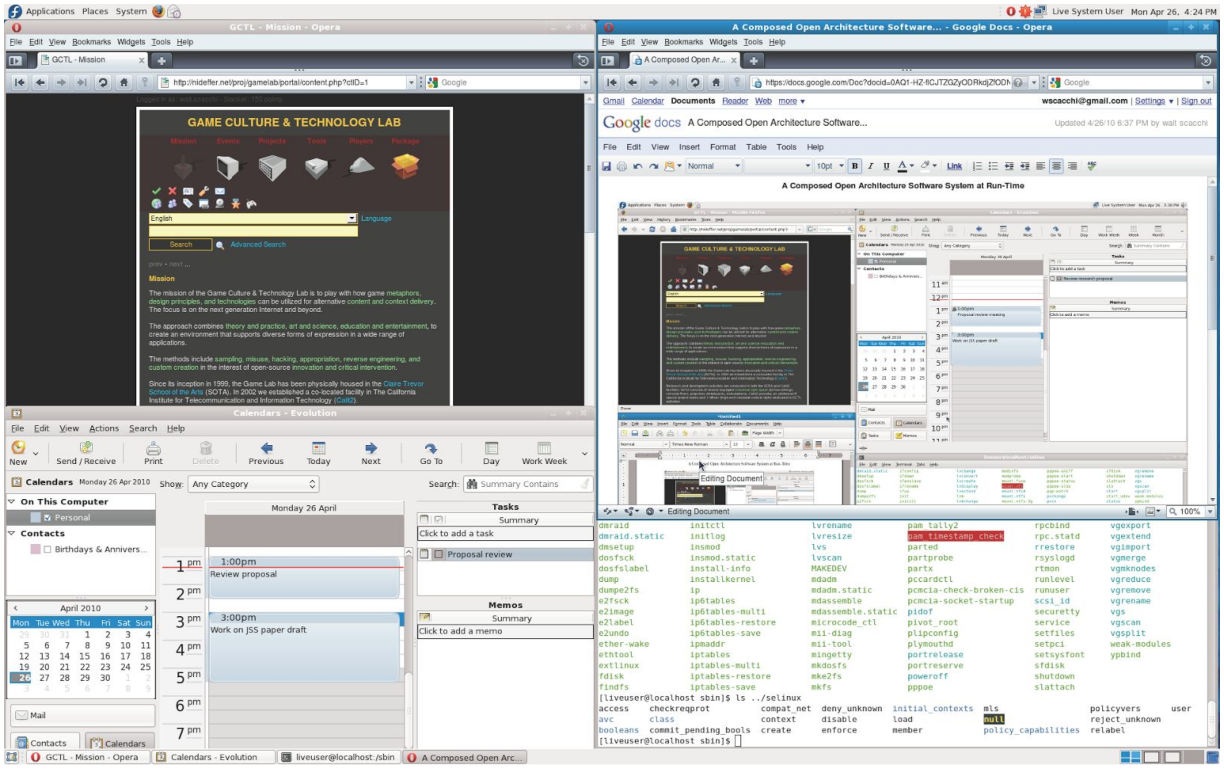
```
dmraid.static     ifconfig      lvchange      modinfo           pppoe-sniff     sfdisk       vgremove
dmsetup           ifdown        lvconvert     modprobe          pppoe-start     shutdown     vgrename
dosfsck           ifenslave     lvcreate      mount.fuse        pppoe-status    slattach     vgs
dosfslabel        ifrename      lvdisplay     mount.nfs         pppoe-stop      sln          vgscan
dump              ifup          lvextend      mount.nfs4        ppp-watch       start        vgsplit
dumpe2fs          init          lvm           mount.ntfs        pvchange        start_udev   weak-modules
e2fsck            initctl       lvmchange     mount.ntfs-3g     pvck            status       ypbind
e2image           initlog       lvmdiskscan   mount.ntfs-fuse   pvcreate        stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot          compat_net  deny_unknown  initial_contexts  mls                   policyvers     user
avc       class                 context     disable       load              null                  reject_unknown
booleans  commit_pending_bools  create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$
```

Figure 7. An instantiation at run time (Firefox, AbiWord, Gnome Evolution, Fedora) of the build-time architecture of Figure 6 that determines the ecosystem niche of Figure 10.


A Composed Open Architecture Software System at Run-Time

```
dmraid.static     ifconfig      lvchange      modinfo           pppoe-sniff     sfdisk       vgremove
dmsetup           ifdown        lvconvert     modprobe          pppoe-start     shutdown     vgrename
dosfsck           ifenslave     lvcreate      mount.fuse        pppoe-status    slattach     vgs
dosfslabel        ifrename      lvdisplay     mount.nfs         pppoe-stop      sln          vgscan
dump              ifup          lvextend      mount.nfs4        ppp-watch       start        vgsplit
dumpe2fs          init          lvm           mount.ntfs        pvchange        start_udev   weak-modules
e2fsck            initctl       lvmchange     mount.ntfs-3g     pvck            status       ypbind
e2image           initlog       lvmdiskscan   mount.ntfs-fuse   pvcreate        stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot          compat_net  deny_unknown  initial_contexts  mls                   policyvers     user
avc       class                 context     disable       load              null                  reject_unknown
booleans  commit_pending_bools  create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$
```

Figure 8. A second instantiation at run time (Firefox, Google Docs and Calendar, Fedora) determining the ecosystem niche of Figure 11.


GCTL - Mission - Opera
http://nideffer.net/proj/gamelab/portal/content.php?ctID=1
A Composed Open Architecture Software... - Google Docs - Opera
https://docs.google.com/Doc?docid=0AQ1-HZ-fiCJTZGZyODRkdjZlODN
Calendars - Evolution
Editing Document
100%

GAME CULTURE & TECHNOLOGY LAB

Mission

The mission of the Game Culture & Technology Lab is to play with how game metaphors, design principles, and technologies can be utilized for alternative content and context delivery. The focus is on the next generation Internet and beyond.

The approach combines theory and practice, art and science, education and entertainment, to create an environment that supports diverse forms of expression in a wide range of applications.

The methods include sampling, misuse, hacking, appropriation, reverse engineering, and custom creation in the interest of open-source innovation and critical intervention.

Since its inception in 1999, the Game Lab has been physically housed in the Claire Trevor School of the Arts (SOTA). In 2002 we established a co-located facility in The California Institute for Telecommunication and Information Technology (Calit2).

A Composed Open Architecture Software System at Run-Time

Updated 4/26/10 6:37 PM by walt scacchi

Calendars Monday 26 Apr 2010

Monday 26 April

1:00pm Review proposal

3:00pm Work on JSS paper draft

Tasks: Click to add a task; Proposal review

Memos: Click to add a memo

```
dmraid          initctl              lvrename           pam_tally2                rpcbind       vgexport
dmraid.static   initlog              lvresize           pam_timestamp_check       rpc.statd     vgextend
dmsetup         insmod               lvs                parted                    rrestore      vgimport
dosfsck         insmod.static        lvscan             partprobe                 rsyslogd      vgmerge
dosfslabel      install-info         MAKEDEV            partx                     rtmon         vgmknodes
dump            installkernel        mdadm              pccardctl                 runlevel      vgreduce
dumpe2fs        ip                   mdadm.static       pcmcia-check-broken-cis   runuser       vgremove
e2fsck          ip6tables            mdassemble         pcmcia-socket-startup     scsi_id       vgrename
e2image         ip6tables-multi      mdassemble.static  pidof                     securetty     vgs
e2label         ip6tables-restore    microcode_ctl      pivot_root                service       vgscan
e2undo          ip6tables-save       mii-diag           plipconfig                setfiles      vgsplit
ether-wake      ipmaddr              mii-tool           plymouthd                 setpci        weak-modules
ethtool         iptables             mingetty           portrelease               setsysfont    ypbind
extlinux        iptables-multi       mkdosfs            portreserve               sfdisk
fdisk           iptables-restore     mke2fs             poweroff                  shutdown
findfs          iptables-save        mkfs               pppoe                     slattach
[liveuser@localhost sbin]$ ls ../selinux
access    checkreqprot         compat_net  deny_unknown  initial_contexts  mls                   policyvers     user
avc       class                context     disable       load              null                  reject_unknown
booleans  commit_pending_bools create      enforce       member            policy_capabilities   relabel
[liveuser@localhost sbin]$
```

Figure 9. A third instantiation at run-time (Opera, Google Docs, Gnome Evolution, Fedora) determining yet another niche conforming to the software supply network of Figure 4.

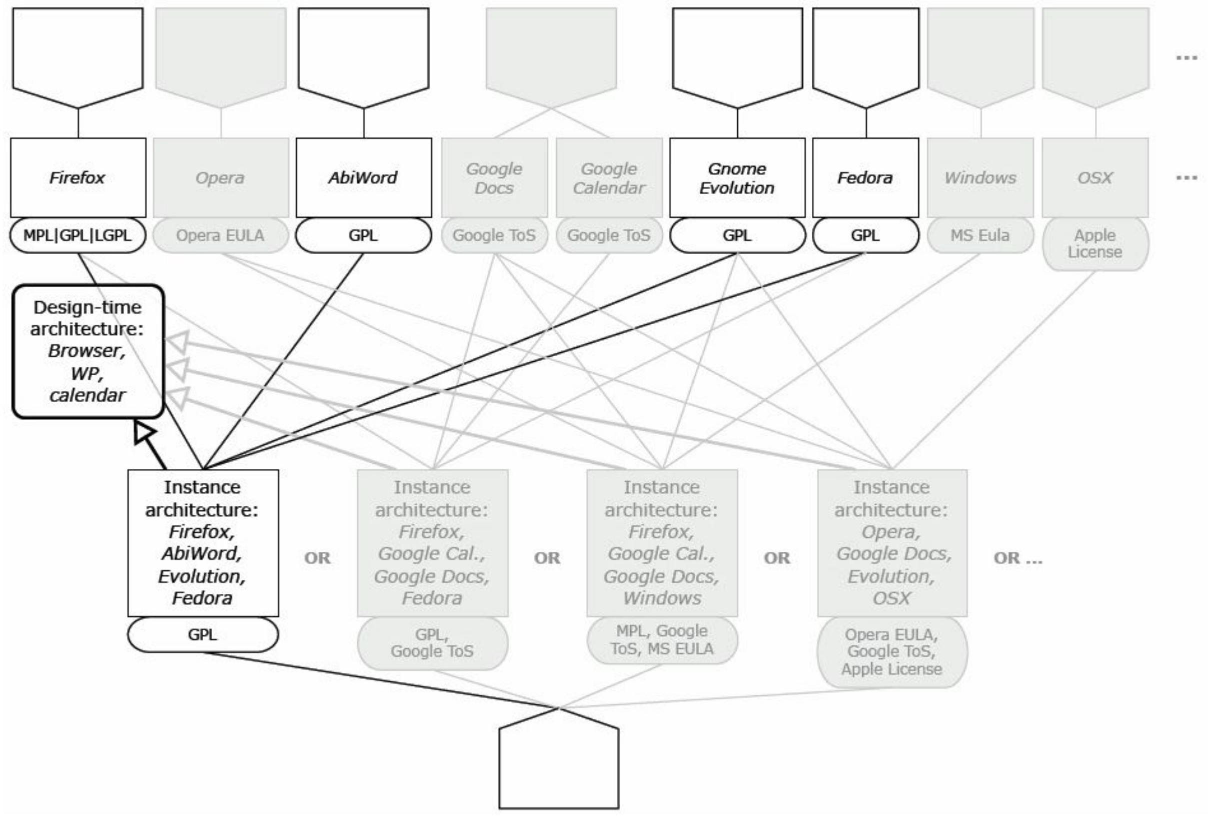

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

Figure 10. The ecosystem niche for one instance architecture.

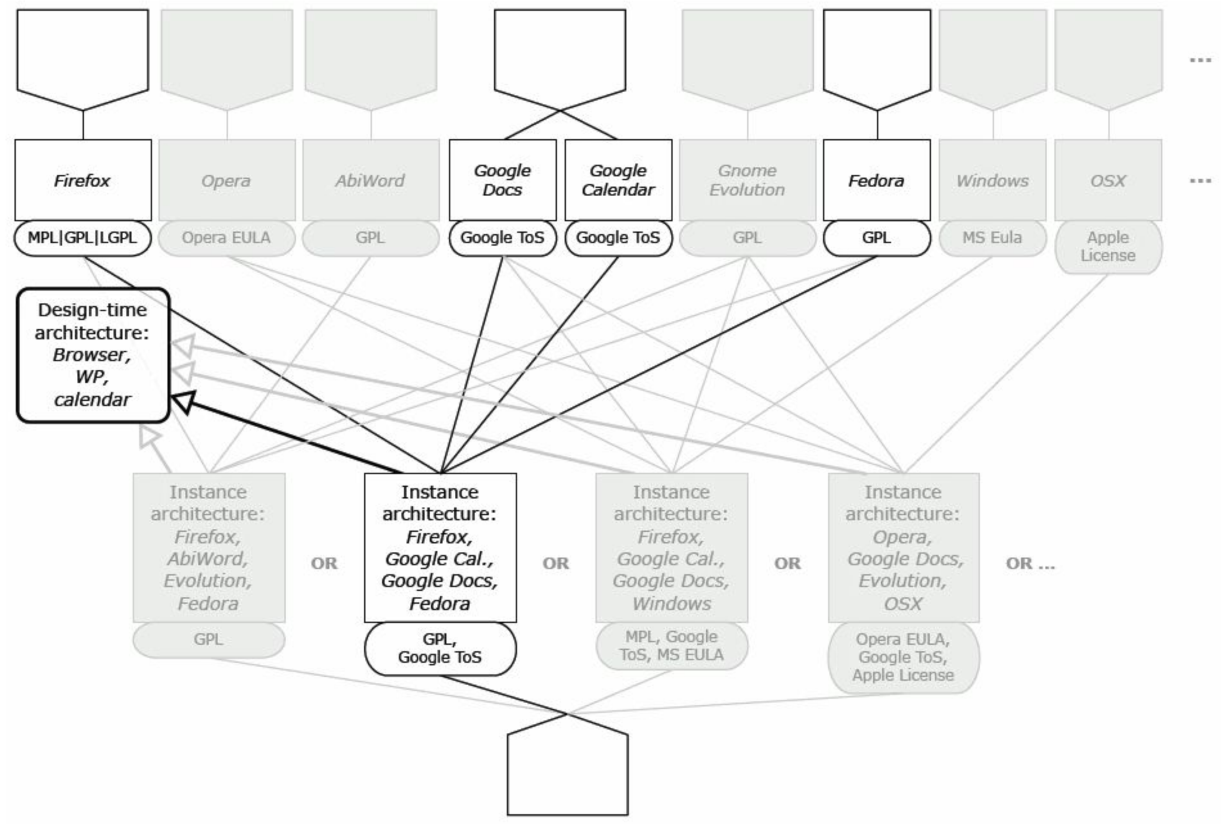

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

Figure 11. The ecosystem niche for a second instance architecture.

This system's ecosystem is complex in important ways:

- Alternatives exist for each component that bring into play diverse possibilities for licenses, evolution paths, system capabilities, requirements, and ecosystems, such as MS Word (proprietary), AbiWord (OSS), or Google Docs (remote service) for the word processor.
- Some component choices co-evolve with coordination among suppliers [such as Mozilla and Gnome components) while others do not.
- The system in its current open architecture is independent of any one supplier. Such ecosystems are more revealing and offer more evolution paths for study (and use) than a system in an ecosystem dominated by a single vendor such as Microsoft, Oracle, or SAP. Single-vendor-dominated ecosystems may be larger, but are less diverse and thus less interesting and offer fewer choices with significant ecosystem impact.
- The system is independent of any one platform; for example, it could be evolved by component replacement to run on a mobile device, moving it into a much different niche.

The system can be instantiated with components all governed by the same license (as in Figure 10), resulting in a monolithically licensed system like Firefox; and it can be instantiated with diversely licensed components (as in Figure 11), resulting in a heterogeneously licensed system like Unity and Google Chrome. Unlike those three, however, it also is an OA system and so its virtual license can be calculated and its software ecosystem niche can be directly studied and evolved toward a more desirable one. Because it is OA, it offers more choices of components and configurations, and thus more possible niches, along with more ways to move among and take advantage of them; all that Firefox, Unity, and Google Chrome offer as expository examples, plus more.

The insights provided by the example system allow one, we believe, to anticipate or even predict the kinds of issues that will arise when new platforms emerge.

The four primary components collectively represent more than a million lines of code. Each component, and its subcomponents recursively down to the smallest, is a composition of other more primitive components that may be independently developed or developed as part of this system, and may be added to the ecosystem relationships in order to consider its effect on supply chains and evolution. An individual component such as Firefox constitutes a micro-platform itself on which Ajax, Rich Internet Applications, or other scripted functionality (e.g. invoking an embedded link to a YouTube Video player) can run internally, constituting an embedded ecosystem. Equivalent components from different OSS or proprietary software producers can be identified, where each alternative is subject to a different type of software license. For example, for Web browsers, we consider the Firefox browser from the Mozilla Foundation, which comes with a choice of OSS license (MPL, GPL, or LGPL), and the Opera browser from Opera Software, which comes with a proprietary software end-user license agreement (EULA). Similarly, for word processor, we consider the OSS AbiWord application (GPL) and Web-based Google Docs service (proprietary Terms of Service).

The OA we describe covers a number of systems we have identified, built, and deployed in a university research laboratory, and as far as can be externally determined also many distinct systems integrated by organizations and distributed internally or to a customer base. We have also developed OA systems with more complex architectures that incorporate components for content management systems (Drupal), wikis (MediaWiki), blogs (B2evolution), teleconferencing and media servers (Flash media server, Red5 media server), chat (BlaB! Lite), Web spiders and search engines (Nutch, Lucene, Sphider), relational database management systems (MySQL), and others. Furthermore, the OSS application stacks and infrastructure (platform) stacks found at BitNami.org/stacks (accessed 29 April 2010) could also be incorporated in OA systems, as could their proprietary counterparts. Even these more complex OAs still reflect the core architectural concepts and constructs, software ecosystem relationships, challenges, and solutions that we present more accessibly in our example system.

The software ecosystem niches for the example system, or indeed any system, depend on which component implementations are used and the architecture in which they are combined and instantiated, as does the overall rights and obligations for the instantiated system. In addition, we build on previous work on heterogeneously licensed systems [ASA10, GeH09, ScA08] by examining how OA development affects and is affected by software ecosystems, and the role of component licenses in shaping OA software ecosystem niches.

Consequently, we focus our attention to understand the ecosystem niche of an open architecture software system:

- It must rest on a license structure of rights and obligations, focusing on obligations that are enactable and testable [AAS09b, ASA10].[2]
- 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 the system's license architecture, the abstraction of its software architecture annotated with licenses, connector types, etc. that determines the system's virtual license (overall rights and obligations) and from which the virtual license can be calculated [AAS09b, AAS11].
- It must account for alternative ways in which software systems, components, and licenses can evolve.
- It must provide an automated environment for creating and managing license architectures. We have developed a prototype that manages the license architecture as a view of the system architecture [AAS09b, AAS11].

Architecture, license, and ecosystem evolution

An OA system can evolve by a number of distinct mechanisms, some of which are common to all systems but others of which are a result of heterogeneous component licenses in a single system. For the application of these mechanisms to systems rather than ecosystems, see our previous work [AAS09a, AAS09b, AAS011, ASA10, ScA08]. By 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 35 to 36). Such minor versions changes generally have no effect on system architecture.

By component replacement— One or more components may be replaced by others with modestly different functionality but similar 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 Writer or MS Word). However, changes in the format or structure of component APIs may necessitate build-time and run-time updates to component connectors. Figure 12 shows some possible alternative system compositions that result from replacing components by others of the same type but with a different license.

[2] 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 General Public License (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?

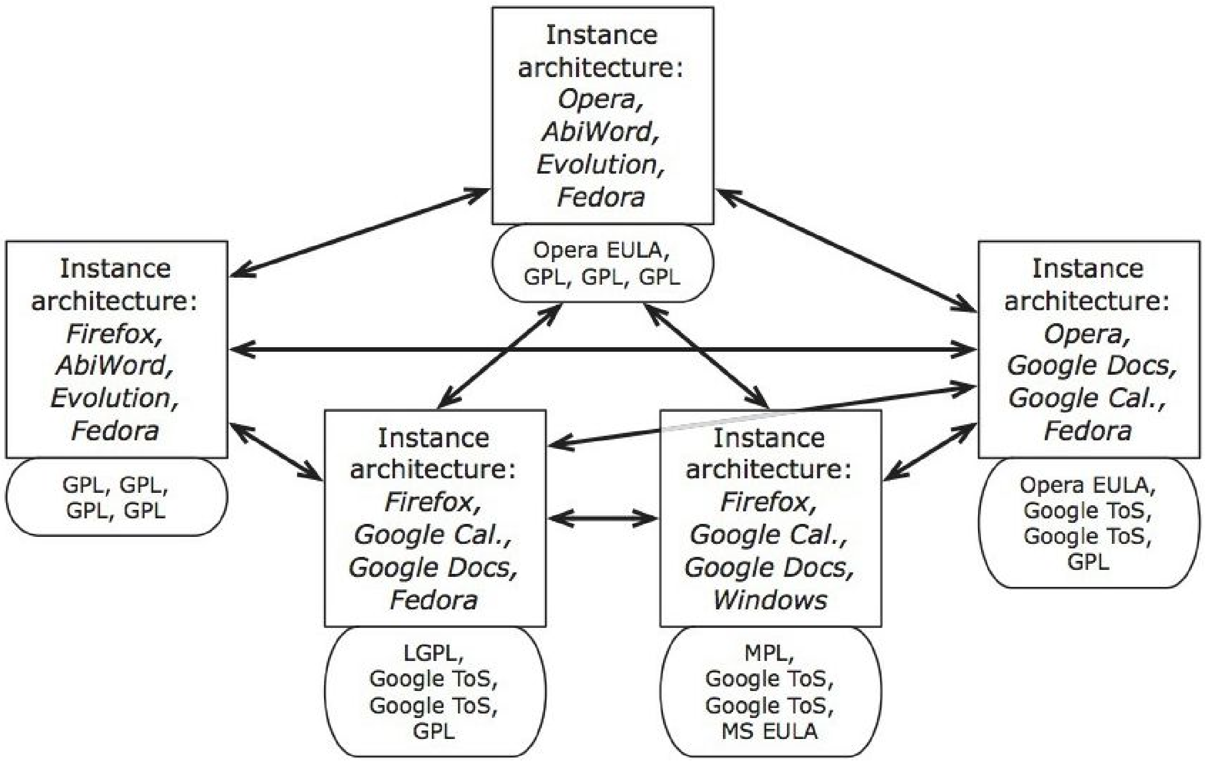
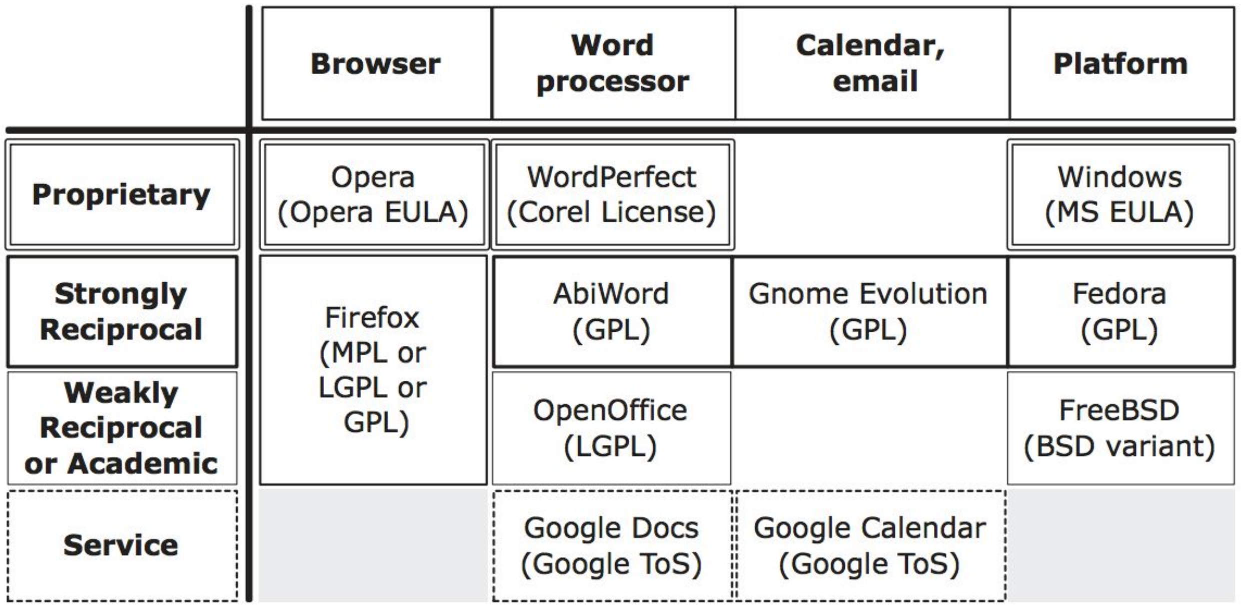

Instance architecture: Opera, AbiWord, Evolution, Fedora
Opera EULA, GPL, GPL, GPL
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL, GPL, GPL, GPL
Instance architecture: Opera, Google Docs, Google Cal., Fedora
Opera EULA, Google ToS, Google ToS, GPL
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
LGPL, Google ToS, Google ToS, GPL
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, Google ToS, MS EULA

Figure 12. Possible evolutionary paths among a few instance architectures; some paths are impractical due to the changes in license obligations

By architecture evolution— The OA can evolve by changing connectors between components rearranging connectors in a different configuration, or changing the interface through which a connector accesses a component, altering the system characteristics. Revising or refactoring the configuration in which a component is connected can change how its license affects the rights and obligations for the overall system. An example is the replacement of word processing, calendaring, email components, and connectors to them with Web-browser-based services such as Google Docs, Google Calendar, and Google Mail. The replacement would eliminate the legacy components and relocate the desired application functionality to operate remotely from within theWeb browser component, resulting in what might be considered a simpler and easier-to-maintain system architecture, but one that is less open and now subject to a proprietary Terms of Service license. System consumer preferences for kinds of licenses and the consequences of subsequent participation in a different ecosystem niche may thus mediate whether such an alternative system architecture is desirable or not.

By component license evolution— The license under which a component is available may change, as for example when the license for the Mozilla core components was changed from the Mozilla Public License (MPL) to the current Mozilla Disjunctive Tri-License; or the component may be made available under a new version of the same license, as for example when the GNU General Public License (GPL) version 3 was released. The three architectures in Figure 12 that incorporate the Firefox Web browser show how its tri-license creates new evolutionary paths by offering different licensing options. These options and paths were not available previously with earlier versions of this component offered under only one or two license alternatives.

In response to different desired rights or acceptable obligations— The OA system's integrator or consumers may desire additional license rights (for example the right to sublicense in addition to the right to distribute), or no longer desire specific rights; or the set of license obligations they find acceptable may change. In either case the OA system evolves, whether by changing components, evolving the architecture, or other means, to provide the desired rights within the scope of the acceptable obligations. For example, they may no longer be willing or able to provide the source code for components within the reciprocality scope of a GPL-licensed module. Figure 13 shows an array of choices among types of licenses for different types of components that appear in the OA example system. Each choice determines the obligations that component producers can demand of their consumers in exchange for the access/usage rights they offer.



	Browser	Word processor	Calendar, email	Platform
Proprietary	Opera (Opera EULA)	WordPerfect (Corel License)		Windows (MS EULA)
Strongly Reciprocal	Firefox (MPL or LGPL or GPL)	AbiWord (GPL)	Gnome Evolution (GPL)	Fedora (GPL)
Weakly Reciprocal or Academic		OpenOffice (LGPL)		FreeBSD (BSD variant)
Service		Google Docs (Google ToS)	Google Calendar (Google ToS)	

Figure 13. Some architecture choices and their license categories.

The interdependence of producers, integrators, and consumers results in a co-evolution of software systems and social networks within an OA ecosystem [Sca07]. 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, as for example is done 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 [AAS09b], license rights and obligations are manifested at each component interface then mediated through the OA of the system to entail the rights and corresponding obligations for the system as a whole. As a result, integrators must frequently re-evaluate the OA system 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

Discussion

At least two topics merit discussion following from our approach to understanding of software ecosystems and ecosystem niches for OA systems: first, how might our results shed light on software systems whose architectures articulate a software product line; and second, what insights might we gain based on the results presented here on possible software license architectures for mobile computing ecosystems. Each is addressed in turn.

Software product lines (SPLs), as introduced in Chapter 2, rely on the development and use of explicit software architectures [Bos00, ClN01]. However, the architecture of an SPL or software ecosystem does not necessarily require an OA—there is no need for it to be open. Thus, we are interested in discussing what happens when SPLs may conform to an OA, and to an OA that may be subject to heterogeneously licensed SPL components. Three considerations come to mind:

- If the SPL is subject to a single homogeneous software license, which may often be the case when a single vendor or government contractor has developed the SPL, then the license may act to reinforce a vendor lock-in situation with its customers. One of the motivating factors for OA is the desire to avoid such lock-in, whether or not the SPL components have open or standards-compliant APIs. However, a single license simplifies determination of the software ecosystem in which these system is located.
- If an OA system employs a reference architecture, then such a reference or design-time architecture effectively defines an SPL consisting of possible different system instantiations composed from similar components from different producers (e.g. different but equivalent Web browsers, word processors, calendaring and email applications). This can be seen in the design-time architecture depicted in Figure 5, the build-time architecture in Figure 6, and the instantiated run-time architectures in Figures 7–9.
- If the SPL is based on an OA that integrates software components from multiple producers or OSS components that are subject to different heterogeneous licenses, then we have the situation analogous to what we have presented in this paper, but now in the form of virtual SPLs from a virtual software production enterprise [NoS99] that spans multiple independent OSS projects and software production enterprises; virtual in the sense that both the enterprise and the SPL are emergent phenomena rather than intended and embodied by existing organizations and business plans. SPL concepts are thus compatible with OA systems that are composed from heterogeneously licensed components, and do not impact the formation or evolution of the software ecosystem niches where such systems may reside.

Our approach for using open software system architectures and component licenses as a lens that focuses attention to certain kinds of relationships within and across software supply networks, software ecosystems, and networks of software ecosystems has yet to be applied to systems on mobile computing platforms. Bosch [Bos09] notes this is a neglected area of study, but one that may offer interesting opportunities for research and software product development. Thus, what happens when we consider Apple iPhone/iPad OS, Google Android OS phones, Nokia Symbian OS phones, Microsoft Windows 7 OS phones, Intel MeeGo/Tizen OS netbooks, or Nintendo DS portable game consoles as possible platforms for OA system design and deployment?

First, all of these devices are just personal computers with operating systems, albeit in small, mobile, and wireless form factors. They represent a mix of mostly proprietary operating system platforms, though some employ Linux-based or other OSS alternative operating systems.

Second, Mobile OS platforms owners (Apple, Nokia, Google, Microsoft) are all acting to control the software ecosystems for consumers of their devices through establishment of logically centralized (but possibly physically decentralized) application distribution repositories or online stores, where the mobile device must invoke a networked link to the repository to acquire (for fee or for free) and install apps. Apple has had the greatest success in this strategy and dominates the global mobile application market and mobile computing software ecosystems. But overall, OA systems are not necessarily excluded from these markets or consumers.

Third, given our design-time architecture of the example system shown in Figure 5, is it possible to identify a build-time version that could produce a run-time version that could be deployed on most or all of these mobile devices? One such build-time architecture would compose an Opera Web browser, with Web services for word processing, calendaring and email, that could be hosted on either proprietary or OSS mobile operating systems. This alternative arises since Opera Software has produced run-time versions of its proprietary Web browser for these mobile operating systems, for accessing the Web via a wireless/cellular phone network connection. Similarly, in Figure 12 the instance architecture on the right could evolve to operate on a mobile platform like an Androidbased mobile device or Symbian-based cell phone. So it appears that mobile computing devices do not pose any unusual challenges for our approach in terms of understanding their software ecosystems or the ecosystem niches for OA systems that could be hosted on such devices.

Conclusion

The role of software ecosystems in the development and evolution of heterogeneously licensed open architecture systems has received insufficient consideration. Such systems are composed of components potentially under two or more licenses, open source software or proprietary or both, in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring. The software licenses of the components both facilitate and constrain in which ecosystems a composed system may lie. In addition, the obligations and rights carried by the licenses are transmitted from the software component producers to system consumers through the architectural choices made by system integrators. Thus software component licenses help determine the contours of the software supply network and software ecosystem niche that emerge for a given implementation of a composed system architecture. Accordingly, we described examples for systems whose host software platform span the range of personal computer operating systems, Web services, and mobile computing devices.

Consequently, software component licenses and the architectural composition of a system determine the software ecosystem niche in which a system resides. Understanding and describing software ecosystem niches is a key contribution of this work. An example system of an open architecture software system that articulates different software supply networks as ecosystem niches was employed to this end. We examined how the architecture and software component licenses of a composed system at design time, build time, and run time helps determine the system's software ecosystem niche, and provides insight for identifying potential evolutionary paths of software system, architecture, and niches. Similarly, we detailed the ways in which a composed system can evolve over time, and how a software system's evolution can change or shift the software ecosystem niche in which the system resides and thus producer–consumers relationships. Then we described how virtual software product lines can exist through the association between open architectures, software component licenses, and software ecosystems.

Finally, in previous work [AAS09b, AAS09c, ASA10] we identified structures for modeling software licenses and the license architecture of a system, and automated support for calculating its rights and obligations. Such capabilities are needed in order to manage and track an OA system's evolution in the context of its ecosystem niche. We have outlined an approach for achieving these structures and support and sketched how they further the goal of reusing and exchanging alternative software components and software architectural compositions. More work remains to be done, but we believe this approach transforms a vexing problem of stating in detail how study of software ecosystems can be tied to core issues in software engineering like software architecture, product lines, component-based reuse, license management, and evolution, into a manageable one for which workable solutions can be obtained.

References

[AAS09a] Alspaugh, T.A., Asuncion, H.U., Scacchi, W., (2009a. Analyzing software licenses in open architecture software systems. In: *2nd International Workshop on Emerging Trends in FLOSS Research and Development* (FLOSS), pp. 1–4.

[AAS09b] Alspaugh, T.A., Asuncion, H.U., Scacchi, W., (2009b. Intellectual property rights requirements for heterogeneously-licensed systems. In: *17th IEEE International Requirements Engineering Conference* (RE'09), pp. 24–33.

[AAS09c] Alspaugh, T.A., Asuncion, H.U., Scacchi, W., (2009c. The role of software licenses in open architecture ecosystems. In: *First International Workshop on Software Ecosystems* (IWSECO-2009), pp. 4–18.

[AAS11] Alspaugh, T.A., Asuncion, H.U., Scacchi, W., (2011. Presenting software license conflicts through argumentation. In: *23rd International Conference on Software Engineering and Knowledge Engineering* (SEKE 2011), pp. 509–514.

[ASA10] 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.

[BCK03] Bass, L., Clements, P., Kazman, R., (2003. *Software Architecture in Practice*. AddisonWesley Longman.

[Bos00] Bosch, J., (2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley.

[Bos09] Bosch, J., (2009. From software product lines to software ecosystems. In: *13th International Software Product Line Conference* (SPLC'09), pp. 111–119.

[BBS10] Bosch, J., Bosch-Sijtsema, P., (2010. From integration to composition: on the impact of software product lines, global development and ecosystems. *Journal of Systems and Software* 83 (1), 67–76.

[BJB09] Boucharas, V., Jansen, S., Brinkkemper, S., (2009. Formalizing software ecosystem modeling. In: *First International Workshop on Open Component Ecosystems* (IWOCE'09), pp. 41–50.

[BrB02] Brown, A.W., Booch, G., (2002. Reusing open-source software and practices: the impact of open-source on commercial vendors. In: *7th. Intern Conf. Software Reuse: Methods, Techniques, and Tools* (ICSR-7), pp. 381–428.

[Chr09] Chromium issues, (2009. *Issue 10638: remove JSCRE from about:credits*. https://code.google.com/p/chromium/issues/detail?id=10638 .

[Chr11] Chromium, (2011. *Chromium terms and conditions*. http://code.google.com/chromium/terms.html

[ClN01] Clements, P., Northrop, L., (2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional.

[Fel07] Feldt, K., (2007. *Programming Firefox: Building Rich Internet Applications with XUL*. O'Reilly Media, Inc.

[Fra10] Franz, M. (2010). E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism, *New Security Paradigms Workshop (NSPW'10)*, Sept. 21–23, Concord, Massachusetts, USA.

[GeH09] German, D.M., Hassan, A.E., (2009. License integration patterns: dealing with licenses mismatches in component-based development. In: *28th International Conference on Software Engineering* (ICSE '09), pp. 188–198.

[Hut57] Hutchinson, G.E., 1957. Concluding remarks. *Cold Spring Harbor Symposia on Quantitative Biology* 22 (2), 415–427.

[JBF09a] Jansen, S., Brinkkemper, S., Finkelstein, A., (2009a. Business network management as a survival strategy: a tale of two software ecosystems. In: *First Workshop on Software Ecosystems*, pp. 34–48.

[JFN09b] Jansen, S., Finkelstein, A., Brinkkemper, S., (2009b. A sense of community: a research agenda for software ecosystems. In: *28th International Conference on Software Engineering* (ICSE ’09), Companion Volume, pp. 187–190.

[JeS05] Jensen, C., Scacchi, W., (2005. Process modeling across the web information infrastructure. *Software Process: Improvement and Practice* 10 (3), 255–272.

[JeS11] Jensen, C., Scacchi, W., (2011. License update and migration processes in open source software projects. In: Hissam, S., Russo, B., de Mendonc¸ a Neto, M., Kon, F. (Eds.), *Open Source Systems: Grounding Research. IFIP Advances in Information and Communication Technology*, pp. 177–195.

[Kue08] Kuehnel, A.-K., (2008. Microsoft, open source and the software ecosystem: of predators and prey – the leopard can change its spots. *Information & Communucation Technology Law* 17 (2), 107–124.

[KWD99] Kuhl, F., Weatherly, R., Dahmann, J., 1999. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall.

[MeS03] Messerschmitt, D.G., Szyperski, C., (2003. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press.

[MeO01] Meyers, B.C., Oberndorf, P., (2001. *Managing Software Acquisition: Open Systems and COTS Products*. Addison-Wesley Professional.

[NaS87] Narayanaswamy, K. and Scacchi, W. (1987) Maintaining Configurations of Evolving Software Systems, *IEEE Trans. Software Engineering*, 13(4), 323-334.

[NeC06] Nelson, L., Churchill, E.F., (2006. Repurposing: techniques for reuse and integration of interactive systems. In: *International Conference on Information Reuse and Integration* (IRI-08), 490-495.

[NoS99] Noll, J., Scacchi, W., 1999. Supporting software development in virtual enterprises. *Journal of Digital Information* 1 (4).

[NoS01] Noll, J., Scacchi, W., (2001. Specifying process-oriented hypertext for organizational computing. *Journal of Network and Computing Applications* 24 (1), 39–61.

[OSI11] Open Source Initiative, (2011. *Open Source Definition*. http://www.opensource.org/docs/osd .

[Ore00] Oreizy, P., (2000. *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution*. PhD Thesis, University of California, Irvine.

[ORM03] Ovaska, P., Rossi, M., Marttiin, P., (2003. Architecture as a coordination tool in multisite software development. *Software Process: Improvement and Practice* 8 (4), 233–247.

[RBC12] Reed, H., Benito, P., Collens, J., and Stein, F. (2012). Supporting Agile C2 with an Agile and Adaptive IT Ecosystem, *Proc. 17th Intern. Command and Control Research and Technology Symposium* (ICCRTS), Paper-044, Fairfax, VA, June 2012.

[Ros05] Rosen, L., (2005. *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall.

[Sca07] Scacchi, W., (2007. Free/open source software development: recent research results and emerging opportunities. In: *6th Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC/FSE 2007), pp. 459–468.

[ScA08] Scacchi, W., Alspaugh, T.A., (2008. Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In: *5th Annual Acquisition Research Symposium*, pp. 230–214.

[SJW11] Salamat, B., Jackson, T., Wagner, G., Wimmer, C., Franz, M. (2011). Run-Time Defense against Code Injection Attacks using Replicated Execution, *IEEE Transactions on Dependable and Secure Computing, Volume 8, No. 4*, July 2011.

[SWZ12] Sun, K., Wang,J., Zhang, F. and Stavrou, A. (2012). SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes. *Proc. 19th. Annual Network and Distributed System Security Symposium.*

[TMD09] Taylor, R.N., Medvidovic, N., Dashofy, E.M., (2009. *Software Architecture: Foundations, Theory, and Practice*. Wiley.

[UnT08] Unity Technologies, December (2008. *End User License Agreement.* http://unity3d.com/unity/unity-end-user-license-2.x.html .

[vPB10] van Gurp, J., Prehofer, C., Bosch, J., (2010. Comparing practices for reuse in integration oriented software product lines and large open source software projects. *Software – Practice & Experience* 40 (4), 285–312.

[VeM08] 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.

Chapter 7.

Processes in Securing Open Architecture Software Systems

Chapter 7.

Processes in Securing Open Architecture Software Systems[1]

ABSTRACT
Our goal is to identify and understand issues that arise in the development and evolution processes for securing open architecture (OA) software systems. OA software systems are those developed with a mix of closed source and open source software components that are configured via an explicit system architectural specification. Such a specification may serve as a reference model or product line model for a family of concurrently sustained OA system versions/variants. We employ a case study focusing on an OA software system whose security must be continually sustained throughout its ongoing development and evolution. We limit our focus to software processes surrounding the architectural design, continuous integration, release deployment, and evolution found in the OA system case study. We also focus on the role automated tools, software development support mechanisms, and development practices play in facilitating or constraining these processes through the case study. Our purpose is to identify issues that impinge on modeling (specification) and integration of these processes, and how automated tools mediate these processes, as emerging research problems areas for the software process research community. Finally, our study is informed by related research found in the prescriptive versus descriptive practice of these processes and tool usage in studies of conventional and open source software development projects.

OVERVIEW

Our goal is to identify and understand issues that arise in the development and evolution processes for securing open architecture (OA) software systems. OA software systems are those developed with a mix of closed source software (CSS) components with open APIs, and open source software (OSS) components, that are configured via an explicit system architectural specification. Such a specification may serve as a reference model or product line model for a family of concurrently sustained OA system versions/variants. We seek to research, develop, and refine new software process concepts, techniques, and tools for continuously assuring the security of large-scale OA software systems composed from software components that include proprietary CSS and non-proprietary/free OSS. In the U.S., Federal government acquisition policy, as well as many leading enterprise IT centers, now encourage the use of CSS and OSS in the development, deployment, and evolution of complex, software-intensive OA systems.

In this paper, we employ a case study focusing on an OA software system whose security must be sustained throughout its ongoing development and evolution. We limit our focus to software processes surrounding the architectural design, continuous integration, release deployment, and evolution found in the OA system case study. To be clear, these processes focus on activities that construct and update configurations of software components, and are not the processes for developing the

[1]An earlier version of this chapter appears in the *International Conf. On Software and Systems Processes* (ICSSP'13), San Francisco, CA, May 2013.

components themselves. The components involved in such OA systems have their own development life cycle, often within development projects that are independent or at arm's length from the effort to develop and evolve an OA system composed from such components.

In our case study, we examine a simple OA enterprise computing system that configures a Web browser (Firefox, Opera, etc.), word processor (AbiWord, Google Docs, etc.), email and calendar component (Gnome Evolution, Gmail, etc.), and operating system (RedHat Linux, RedHat Fedora with SELinux, Microsoft Windows, Apple OSX, SEAndroid, etc.) in conjunction with file, mail, and Web servers (which may be on distributed network servers), in a loosely coupled manner. However, even this simple OA system that we study draws on an ecosystem of diverse software component providers, whose software products can be configured into alternative, functionally similar system configurations that conform to an OA software product family, as indicated in Figure 1. Such a OA system is also a core of more complex, mission-critical command and control systems [Giz11, SBN12]. Additionally, such a system can also be built and deployed for use on a mobile computing platform like a tablet or smartphone. Finally, our OA system can be encapsulated within security capability and enforcement mechanisms (e.g., SELinux capabilities, virtual machine hypervisors) in order to secure the OA system [DIS12, Sma12, USC11, Xen13].

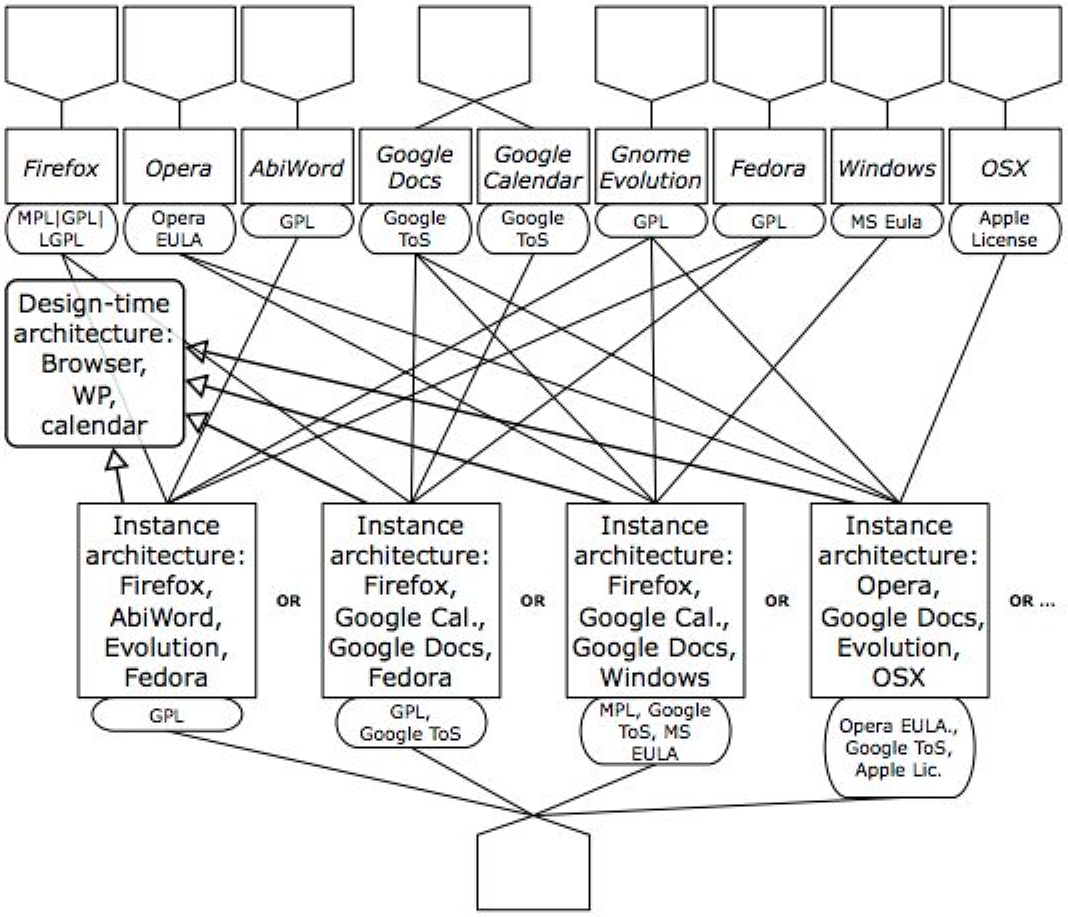

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
OR ...
GPL
GPL, Google ToS
MPL, Google ToS, MS EULA
Opera EULA., Google ToS, Apple Lic.

Figure 1: A software ecosystem of software components that can be configured into a product line indicating four functionally similar OA systems.

We also use the case study to focus on the role automated tools, software development support mechanisms, and development practices play in facilitating or constraining OA software processes. Our purpose is to identify issues impinging on modeling (specifying) and integrating these processes, and explore how automated tools mediate these processes, as emerging research problems areas for the software process research community. We also discuss how such issues affect practical simulation and analysis of these processes.

In the remaining sections of this paper, we first examine related research found in the prescriptive versus descriptive practice of software processes for architectural design, continuous integration, release deployment, and evolution. Next is our case study, describing an OA enterprise computing system that must remain continually secure as it evolves; we use this to help identify issues arising in the specification and integration of the four software processes when the goal of the overall process effort is to continually secure an OA system. We present examples throughout this case study. We then investigate the software process modeling and process integration issues that were observed in this study, as well as how they further constrain efforts to simulate or computationally analyze such processes, and conclude the paper.

RELATED RESEARCH AND DEVELOPMENT EFFORTS

We choose to focus on the processes from architectural design, continuous integration, and release deployment to software evolution for OA systems. Such systems incorporate both CSS and OSS components. In particular, our interest is to examine how these processes enable or constrain how to produce a secure OA system. In particular, we recognized that processes for software architecture design and software evolution [MFP06] have received prior attention in the software process community, but continuous integration and release deployment have received much less attention. Similarly, relatively little is known about how design processes enable and constrain continuous integration and delivery, nor how they in turn facilitate or constrain software evolution. Such an undertaking needs to go beyond prior efforts to specify and identify issues that may arise in processes for the development of component-based software systems [CCL06, QuH08]. Earlier process studies like these do not address, for example, how new development technologies such as continuous integration systems mediate development processes for component-based systems. They also do not identify continuous integration, or software release delivery and installation, as salient development processes for component-based software systems. This may be so as continuous integration and release management are relatively new software development processes, and such processes seem to be visibly practiced in large OSS development projects. Finally, these earlier studies offer little insight as to how functional or non-functional requirements for securing an OA system mediate its software development and evolution processes. But we do know some things about these processes from related efforts, especially for continuous integration.

Continuous integration (CI) systems support automated processes for building, testing, and packaging a software system for release [DMG07, Fow00, Wik12]. Without a CI system, developers must build, test, and integrate their software (component) products using hand-crafted scripts, and it is common for such scripts to have to rely on idiosyncratic dependencies on tool chains and libraries versions for each deployment platform targeted (e.g., [Hyp13]). In contrast, CI systems incorporate the capabilities of software build systems [Smi11] that may invoke sequential, distributed, or parallel builds across multiple build servers (cf. [TCI12]) to produce singular builds (e.g., “nightly builds”), continuously updated agile development builds [Fow00], or diverse, functionally equivalent executable variants [JSH11]. The build systems access and update software code (version control) repositories via process automation scripts. CI sub-processes take as input directories/folders of source code files and produce software component executables. The executables may also be organized as a structured collection (an information architecture) of binary files, static data value and parameter setting files packaged in interlinked directories, constituting releases for deployment. Continuous delivery (CD) further extends CI to support automated release management and the creation of automated deployment tools such as “installation wizards” to be used by system administrators or end-users [HuF10]. For the remainder of our paper, we use the abbreviations CI and CD to refer to these sets of automatable software development processes.

As Fowler [Fow00] observed about the need for continuous integration as an enabling mechanism for agile development, “the key is to automate absolutely everything and run the process so often that integration errors are found quickly. As a result everyone is more prepared to change things when they need to, because they know that if they do cause an integration error, it’s easy to find and fix” (emphasis added). CI processes can therefore be viewed with the assumption that errors resulting from process automation are normal, expected, and not necessarily easily anticipated. But why do these errors occur at all, and why do we need to run the process often in order to identify and resolve integration problems? We need to make closer, systematic observations to determine why or how these errors occur, so that we can advance our process engineering knowledge, as well as to enable practical process improvement. A case study can serve as a starting point for this, and this is our strategy.

Automated CI systems comprise composed environments of software tools, or sets of loosely coupled tools together by automated process invocation scripts that guide and constrain their use. Often these tools are independently developed and evolved. For example, a CI system like Hudson [Hud11] includes source code build tools like Ant or Maven, an issue tracking (or bug reporting) tool like Bugzilla [JeS05] or Jira, and a software revision control browser and search engine like FishEye or ViewVC for viewing the contents of software revision control code repositories like CVS or Subversion. All of these tools happen to be OSS associated with active OSS development projects, so these tools are subject to ongoing development and evolution that improve their capabilities and add/remove functionality. Other CI systems may use different tools or locally developed capabilities in place of external OSS tools such as these. Consequently, this implies the process steps enacted by a CI system will vary (and evolve) depending on the choice of CI system, and on the external tools or locally embedded software functionality that particular CI system uses. Whether such CI process steps are equivalent, similar, or incongruent across CI systems thus remains an open issue. But it is an issue that must be resolved when transitioning from one CI system, or CI system version, to another. However, current CI systems do not appear to address this, nor do they identify it as a concern in their recommended best practices (cf. [Hud11, TCI12]). Similarly, when we add the need to address the CI and CD of secure OA systems, we quickly finds gaps in the best practices that point to shortfalls either on the CI/CD process support side, the security capability side [USC11], or their interdependencies.

Automated CI systems are continuously being improved or supplanted [Jen13, Kri11] and different CI systems offer different features, functional capabilities, and depend on different software tools [TCI12].

The same can be said for CD/release deployment systems, especially with regard to ongoing advances and refinement of software packagers, file distribution and mirror (copy server) synchronization, installers, and uninstallers [HuF10]. So from a software process specification or modeling viewpoint, there are many distinct CI process instance types, and no single abstract CI or release deployment process prescription to follow and tailor to local development organization needs. CI and CD process enactment must therefore rely on manual best practices in addition to tool-based automation, and these practices are specific to each CI system and the tools therein [Hud11]. CI and release management system-based process automation thus is both ad hoc and idiosyncratic, rather than easily standardized or generalized, yet is a widespread software engineering process and practice used to produce thousands of software components (e.g., smartphone or tablet apps).

Software delivery and deployment suffer similar kinds of process automation pathologies (e.g., [IBM07]), to the extent that a key advantage of automation is now thought to be finding or process enactment errors, mistakes, or other articulation problems [MiS91] by running the enactment more quickly. Software deployment errors, such as releasing and installing a premature system release candidate into production operations can have devastating technical or economic consequences, as was demonstrated by the experience of Knight Capital in Summer 2012 [Dig12]. How to provide automated tools and practical techniques that provide (more) robust acceptance/compliance checking prior to a new system version being installed prior to going live in operation, seems to be an underspecified process enactment problem. Adding robust diversity mechanisms and capabilities for dramatically improving OA system security [GST2, JSH11, ScA13] remains an open question for further study. Once again, a case study can serve as a starting point for examining such issues and concerns, and this is our strategy. We see that part of the process challenge is how to understand and specify software processes that must interface with emerging CI and CD systems. These CI systems entail different kinds with different build, package, and release deployment process automation capabilities, or that produce integrated systems that operate on different platforms [TCI12]. To us, this raises concerns for process specification— determining what aspects of a software process are pertinent for modeling and simulation, as well as contributory to improving process effectiveness [RKA11], and process integration— integrating modeled process specifications with diverse automated process enactment mechanisms [MiS92]. It also raises issues for integration across multiple process representations that are supported by independently developed, heterogeneous process enactment mechanisms [GPS94].

CASE STUDY: *A SECURE OA ENTERPRISE SYSTEM*

We utilize a case study to explore and identify software process issues that arise while producing a secure enterprise computing software system. Such a system is produced using existing software applications as components, composing and configuring them to realize the overall system. The processes we examine are not those that develop such software applications, but rather those that use them as components of the system. However, this choice still highlights how the ongoing, independent development and evolution of the components motivates new versions/variants of the overall OA system. In this regard, software component evolution is a driving force that impinges on the development and evolution of OA systems incorporating such components.

Another aspect of our study is to recognize some software processes, like architectural design and software evolution, as having limited automated enactment, while others such as continuous integration and release management are potentially fully automated. This is not to say that no tools are involved in design or evolution, far from it. Rather, what is of interest is that software production and system integration organizations employ a flow of software processes that employ both fully and partially automated enactment. Assuming a world where all software processes are fully automated may be another challenge, but it is not one that is of practical use or consequence at this time. Our study thus addresses software process challenges that are both reflective of understanding of emerging software process research issues, and also may have practical application today and beyond. As such, we turn to our case study to elaborate the software processes of interest, and to the issues they raise for software process research.

Architectural Design Process

The process for designing the configuration of an OA system at the component level is our focus here. We start by noting that we assume no pre-existing process model or standard for such a process, nor do we propose to provide such a prescriptive process. As a review of the architectures of dozens of OSS systems [BrW12] makes clear, there is no common prescriptive process, preferred set of tools, nor is there notational scheme for the architectural design of open software systems. Instead, we describe aspects of a design process we developed, practiced, and adapted that is supported in part with automated design tools. One of our goals with this process was to help identify situations, and practical nonfunctional requirements, that arise with an OA design process that constrains, and is constrained by, the other three downstream software processes in our study.

We have used an OA tailored version of the UCI ArchStudio4 architecture design system (oAS4) as a locally developed plug-in to the Eclipse IDE to realize a partially automated system for architectural design activities [AAS12, AAS13]. oAS4 allows us to visually model the architectural configuration of software components, component interfaces, and component connectors as OA system elements. oAS4 also produces output in an architectural description language (ADL) as a persistent artifact for external analysis, or for potential integration with CI systems with further processing (e.g., binding component classes to their build-time instances). We further focus our architectural design activities to produce an abstract system architecture that serves to denote a product line model of a family of alternative system configurations composed from functionally similar components or component versions [ScA12]. oAS4 can thus support our experimental studies in OA system design and design evolution across families of alternative system configurations (cf. an earlier approach to such problems at [NaS97]).

We annotate our OA system designs within oAS4 using formal constraint expressions on components interfaces, such as intellectual property (IP) license obligations and rights [AAS12, AAS13]. Security policy constraints for components, configured sub-systems, or an overall system are expressed and analyzed in a similar manner [ScA13]. The ability to model and automatically analyze such obligations and rights is needed at build-time and release deployment-time. Automated analysis mechanisms then allow us to determine whether the specified component interconnections entail matches or conflicts in component-component license alignments [AAS12, AAS13]. However, we have also observed that design-time actions must accommodate build-time and deployment-time element bindings, as well as accommodate the evolution of licenses, policies, and system element versions [ScA12]. For example, when con- flicts are found between the licenses of interconnected build-time component selections, we can then reconfigure our OA system design to eliminate the conflicts, to constrain the selection of components at build-time (within CI) to those whose licenses will match or not conflict, or to wrap/shim a component with an abstraction layer that does not transfer IP license obligations.

Design of OA systems also raises issues for how to how best to secure the designed system architecture [USC11]. Among the recommended practices for designing secure system architectures are to provide capability-based user/developer access control that effectively limits access to input and output data, internal program code representations (e.g., memory address and system name spaces), persistent data storage, and to exposed I/O transaction processing interfaces. One increasingly common approach is to provide encapsulation mechanisms like virtual machines for software components or (sub-)system configurations, along with encrypted inter-component data/control flow connectors (e.g., HTTPS/SSL data communication protocols). Of these, passively secure connectors for networked components are widely available, while dynamically secured connectors are a recent advance [GST12]. In our case, we choose to incorporate virtual machines to encapsulate our OA system, and we ignore alternative security protection schemes for simplicity. However, we recognized that even a seemingly simple decision like this still requires analyzing trade-offs about whether to encapsulate the entire system as a single virtual machine (relatively easy to address during deployment, though requiring deployment and installation of virtual machine software (e.g., [Xen13]) on the target deployment computers) or to encapsulate each different component within its own virtual machine that would then be interconnected using secure connectors (more challenging to address for deployment, but offering a more resilient OA system security [ScA13]. We decided to design something in-between these two extremes, by taking into account where different components might be hosted within a networked, multi-server platform environment. What our OA system design process produced is an abstract architectural configuration of component types (each attributed with IP license constraints—not shown but described elsewhere [AAS12, AAS13, ScA13]), a minimal component interconnection scheme, and what we call a hybrid virtual machine confinement scheme, as shown in Figure 2.

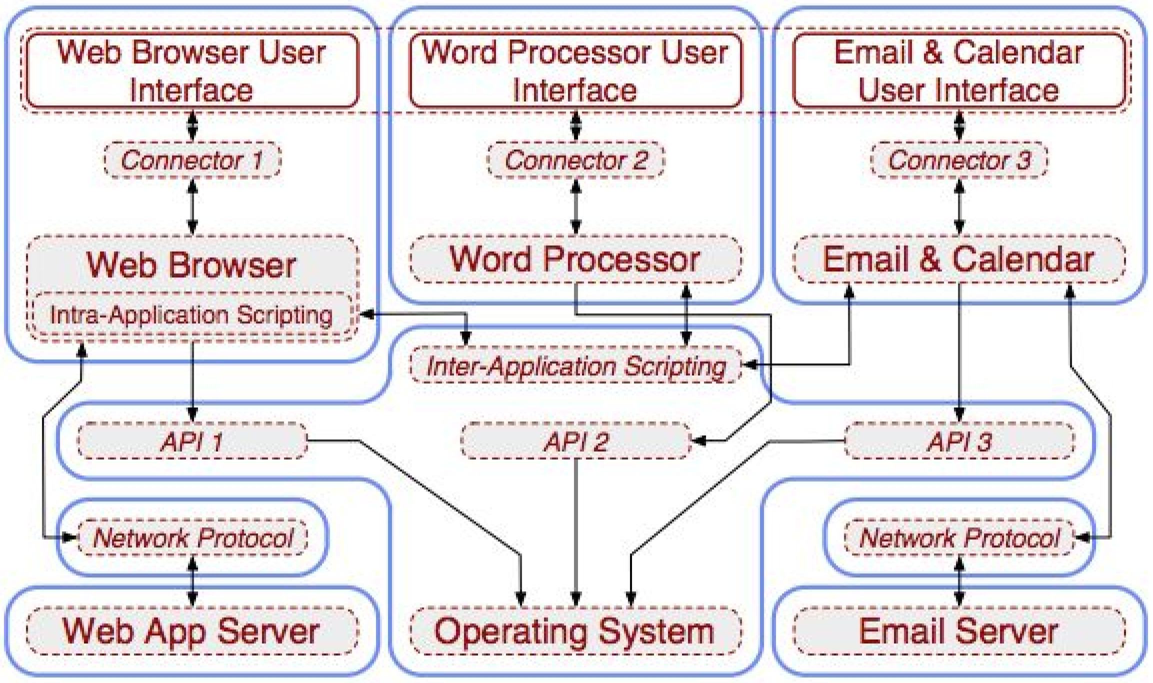


Figure 2: Design configuration of a secure OA enterprise system, shown with a security encapsulation layout. Other encapsulation schemes are possible.

Given that we have so far only examined the architectural design process, we note that we are already beginning to see that we need to anticipate non-functional requirements for the other downstream software processes that follow, particularly in the form of process enactment directives or constraints. We also begin to anticipate whether such information can be automatically propagated into the process automation tools used in these downstream processes.

Continuous Integration Process

In our study, one of the first activities in moving from architectural design to continuous integration is to identify specific software component versions that can be instantiated within the current architectural configuration (Figure 2). While at first it might seem that this is a simple task, we have found that component and version selection are subject to the obligations and rights stipulated with a component's associated IP license [AAS12]. For example, common architectural design languages do not specify annotations for IP licenses, so as noted above, we extended our ADL within the oAS4 with IP obligation and right constraints [AAS12, AAS13]. This meant we could now analyze whether or how IP obligations and rights for each component-component interconnection match, conflict, or propagate. For example, reciprocal licenses like GPL can propagate their IP regime by design, though some enterprises seek to avoid this. By conceptually filling in selected component licenses, we can tell, prior to integration, whether the resulting release candidate may suffer from licensing problems or not. When conflicts or mis-matches are discovered, again prior to further build-time process actions, alternative components with the similar functional capabilities and interfaces but different licenses may be substituted. Alternatively, the architectural configuration can be modified, for example, wrapping a component in a way that mitigates license conflicts (e.g., replacing a direct API-API interconnection which propagates license restrictions with an networked data communications link, as few licenses propagate IP across network connections).

What we end up with from our build sub-process is a concrete OA system configuration with a specific selection of software components specified using oAS4, whose output is intended for a manual build system or for entry into an automated CI system. A concrete configuration is seen in Figure 3. So our build sub-process can now instantiate components into a reusable OA software product line design, as we can determine families of component version instances that can be substituted within the OA system. For example, the Firefox Web browser may be replaced by Google Chrome in this configuration, because both are under permissive OSS licenses. However, a license match/conflict assessment would be required before replacing Firefox with Microsoft Internet Explorer (IE) or Opera, each of which is under a proprietary license. But in the abstract and concrete architectural configuration we have, we could substitute a Linux-based Opera browser without issue, but not IE, unless we add a library wrapper such as Wine [Win13], in order to run IE on Fedora Linux.

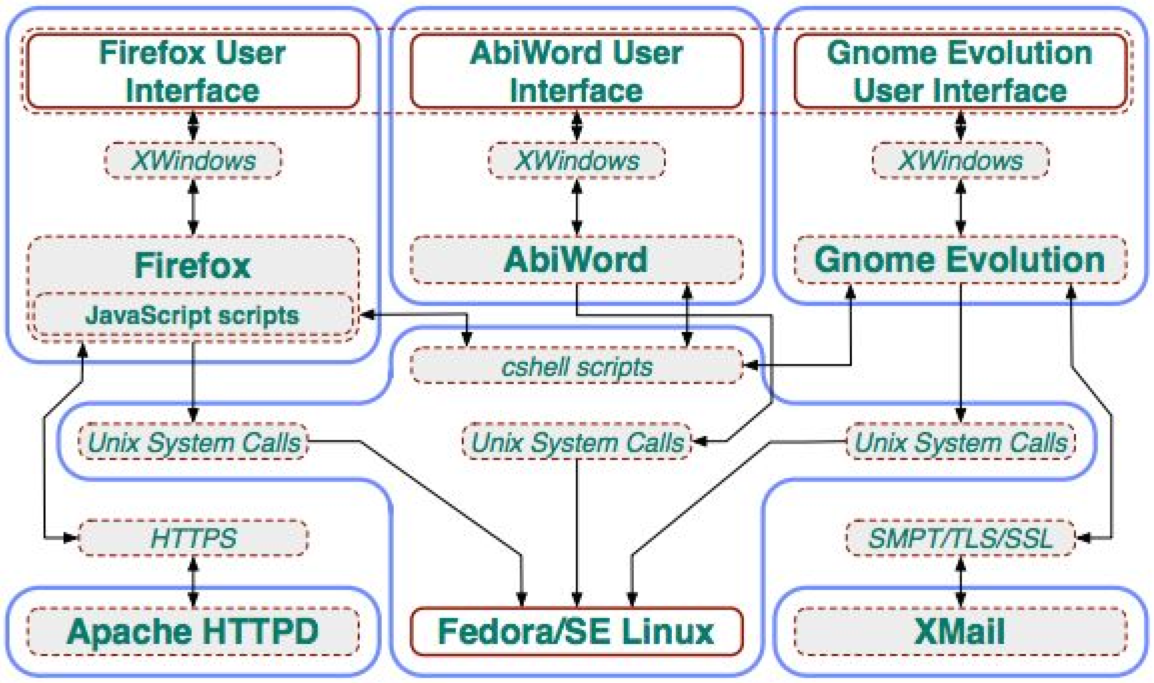


Figure 3: An integration and test build-time configuration of a secure OA enterprise computing system, following the design in Figure 2.

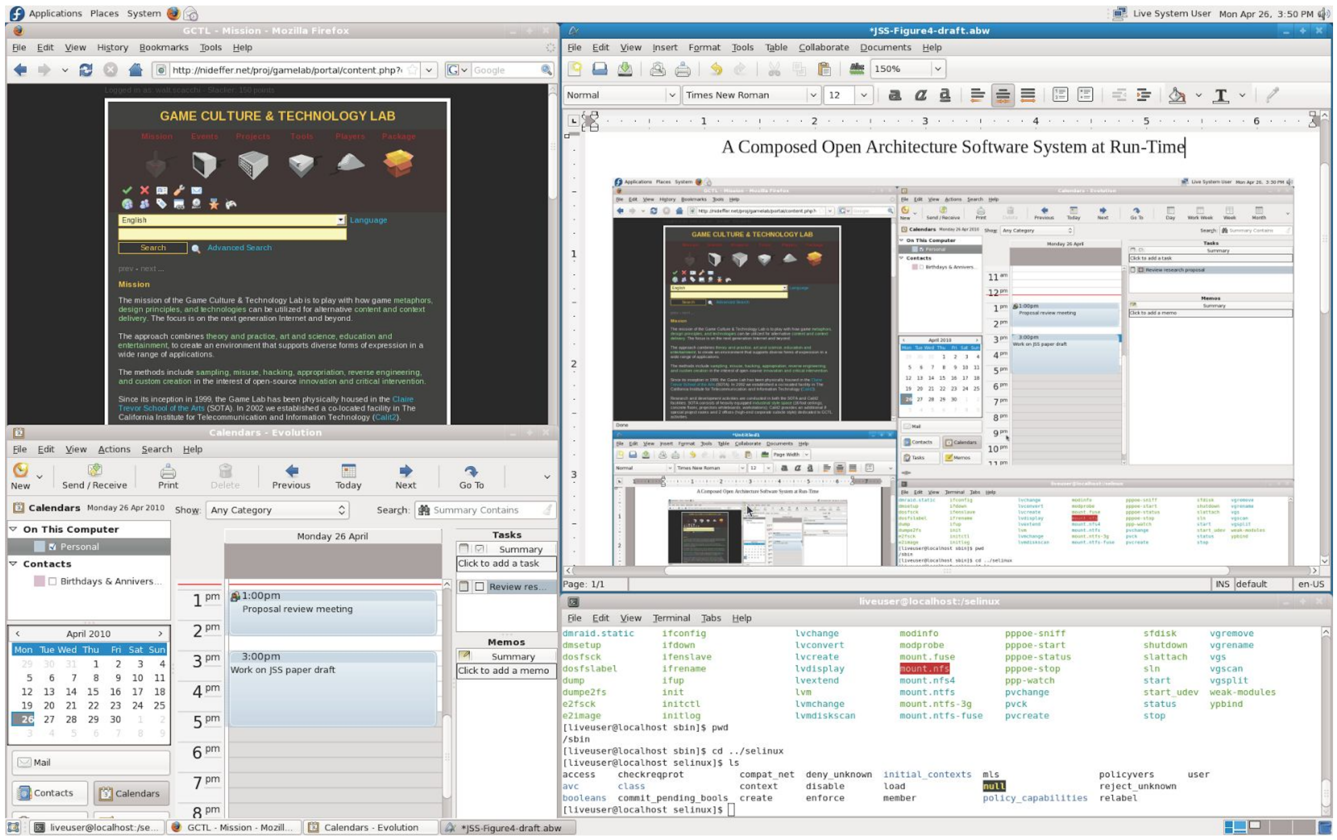
So far, so good. But now we must consider how to transfer this component selection specification into the build system arises. An ideal solution might involve an automated hand-off. However, the specifics of such a hand-off will vary depending on the build system and the CI system we select. A more general solution would likely require (or benefit from) another abstraction layer for integration between the architectural design and build/CI process enactment mechanisms, which is an already recognized problem with a demonstrable solution (cf. [GPS94]). We see that software process research may demonstrate solutions to messy process integration issues, but integration of process flows across toolspecific process enactment representations and automated mechanisms remains a lingering, practical problem that is not yet addressed by current CI or CD systems.

A similar problem arises when we consider how to secure the concrete OA system configuration. For example, we can choose to include secure data communication connectors (e.g. secure protocols like

HTTPS and TLS/SSL) in our configuration, but such capabilities are not instantiated at build-time. Instead, they depend on mechanisms and data (e.g., certificates) that are accessed at run-time once an integrated system release candidate is available. An OA system, or OA system components, can also be secured using virtual machine hypervisors [Xen13] that confine and isolate deployed system/component within a virtual machine run-time environment. In addition, it should be possible to specify operating system access control and type enforcement capabilities (e.g., using SELinux libraries on Fedora), but again, these are not available for use until there is a deployable integrated system release candidate. Thus, these forms of security are most likely invisible to current CI systems, and must be addressed through other means.

Release Deployment Process

The software system you release and deploy depends on what (and how) you build and package for release and installation. For example, in our enterprise system, we want our software integration process to produce a run-time version of our designed software configuration for our target platform (e.g., local personal computer). Figure 4 displays a run-time instantiation in operation, based on the build-time configuration in Figure 3, hosted on a Fedora Linux operating system that utilizes the SELinux library to set access control and run-time capabilities for files and programs.



Applications Places System
Live System User Mon Apr 26, 3:50 PM
GCTL - Mission - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://nideffer.net/proj/gamelab/portal/content.php?
GAME CULTURE & TECHNOLOGY LAB
English
Language
Search
Advanced Search
prev - next ...
*JSS-Figure4-draft.abw
File Edit View Insert Format Tools Table Collaborate Documents Help
150%
Normal
Times New Roman
12
Page: 1/1
INS
default
en-US
Calendars - Evolution
File Edit View Actions Search Help
New
Send / Receive
Print
Delete
Previous
Today
Next
Go To
Calendars Monday 26 Apr 2010
Show: Any Category
Search: Summary Contains
On This Computer
Personal
Contacts
Birthdays & Annivers...
Monday 26 April
Tasks
Summary
Click to add a task
Review res...
Memos
Summary
Click to add a memo
April 2010
Mail
Contacts
Calendars
liveuser@localhost:/selinux
File Edit View Terminal Tabs Help
liveuser@localhost:/se...
GCTL - Mission - Mozill...
Calendars - Evolution
*JSS-Figure4-draft.abw

A Composed Open Architecture Software System at Run-Time

Mission

The mission of the Game Culture & Technology Lab is to play with how game metaphors, design principles, and technologies can be utilized for alternative content and context delivery. The focus is on the next generation Internet and beyond.

The approach combines theory and practice, art and science, education and entertainment, to create an environment that supports diverse forms of expression in a wide range of applications.

The methods include sampling, misuse, hacking, appropriation, reverse engineering, and custom creation in the interest of open-source innovation and critical intervention.

Since its inception in 1999, the Game Lab has been physically housed in the Claire Trevor School of the Arts (SOTA). In 2002 we established a co-located facility in The California Institute for Telecommunication and Information Technology (Calit2).

1:00pm Proposal review meeting

3:00pm Work on JSS paper draft

```
dmraid.static   ifconfig        lvchange        modinfo         pppoe-sniff     sfdisk      vgremove
dmsetup         ifdown          lvconvert       modprobe        pppoe-start     shutdown    vgrename
dosfsck         ifenslave       lvcreate        mount.fuse      pppoe-status    slattach    vgs
dosfslabel      ifrename        lvdisplay       mount.nfs       pppoe-stop      sln         vgscan
dump            ifup            lvextend        mount.nfs4      ppp-watch       start       vgsplit
dumpe2fs        init            lvm             mount.ntfs      pvchange        start_udev  weak-modules
e2fsck          initctl         lvmchange       mount.ntfs-3g   pvck            status      ypbind
e2image         initlog         lvmdiskscan     mount.ntfs-fuse pvcreate        stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access   checkreqprot         compat_net  deny_unknown  initial_contexts  mls                   policyvers     user
avc      class                context     disable       load              null                  reject_unknown
booleans commit_pending_bools create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$ 
```

Figure 4: A screenshot view of a deployed release configuration of our OA enterprise computing system.

However, what we build and what we release may not be the same, though they need to be functionally equivalent. For example, when we select one or more CSS components (an already compiled and integrated executable binary image) with a common restrictive IP license (i.e, one that prohibits copying or redistribution) for inclusion in our build-time architectural configuration, during the build process, we must link it as an executable binary for inclusion in a release candidate for deployment (or deployment testing) (cf. [JeS05]) on a local computer. Such inclusion is a prerequisite for overall integrated system testing processes required by CI. Nonetheless, we cannot distribute such a release candidate to others, as it is common for CSS to not allow duplication or distribution of licensed copies of software binaries. Instead, we need to specify and configure a deployment-platform specific automated software installation mechanism (e.g., installation wizard) that needs to search for and find a local licensed copy of the CSS executable binary, and link it to the result of the build sub-process that provides a run-time linkage mechanism in expectation. A similar effort is needed to enable user acceptance testing or certification testing on their local platform. These release deployment process steps can be accomplished with some effort, but this effort could also be anticipated at design-time or build-time, when developers make their selection for which component instances to include in the system build.

Automated software installation is an increasingly common expectation. Software installers run automated process enactment scripts crafted by developers. Once again, the process being enacted is not explicitly specified, nor is it separate from the internally coded software utility's action invocation scripts. This means that it is not surprising to discover errors that arise during installation but are not easily anticipated without extensive prior experience in working with the installer on known target platforms. For example, an informal aid from IBM for guiding system administrators who enact software installation processes [IBM07] notes installation problems like: (a) insufficient free space on disk storage prior to or during software executable installation; (b) software installations across a network that are "hung" or stuck due to lack of robust installation protocols that can time-out (abort) and/or re-initiate then re-validate process script commands already invoked; (c) installations that fail due to underspecified all/nothing installation transactions (cf. [Gra81]) that do not completely update the information architecture of a multi-part software configuration (e.g., program registry update and reversible roll-back to prior registry; and/or setup of user configuration files); (d) failure to include a software uninstaller (or uninstallation process scripts) that allows conditional roll-back to previously installed software versions to be retrieved and activated; or (e) file/directory name collisions that arise at build-time versus deployment-time.

Our observation is that if there is a sufficiently detailed, informing process specification or model for how best to install a software release, it is well hidden. We all rely on the correct operation and outcome on software installation processes on our networked personal computers and wireless mobile devices, but such processes often are problematic or fail. This situation is not inevitable, but it is widespread. There is a missed opportunity to improve the quality of release deployment process outcomes by some means other than the costly software installation trial and error learning experiences that afflict software release deployment personnel and system administrators. We should be able to do much better than this. The provision of explicit software installation process models that can guide the targeting of different deployment platforms in specific organizations or for remote users begs for research and development attention.

Evolution Process

An OA system can evolve by a number of distinct mechanisms or process enactment pathways, some of which are common to all systems, but others of which arise only in OA systems or where components in a single system are heterogeneously licensed [AAS13]. Figure 5 provides a summary of some of the various paths, further explained below.

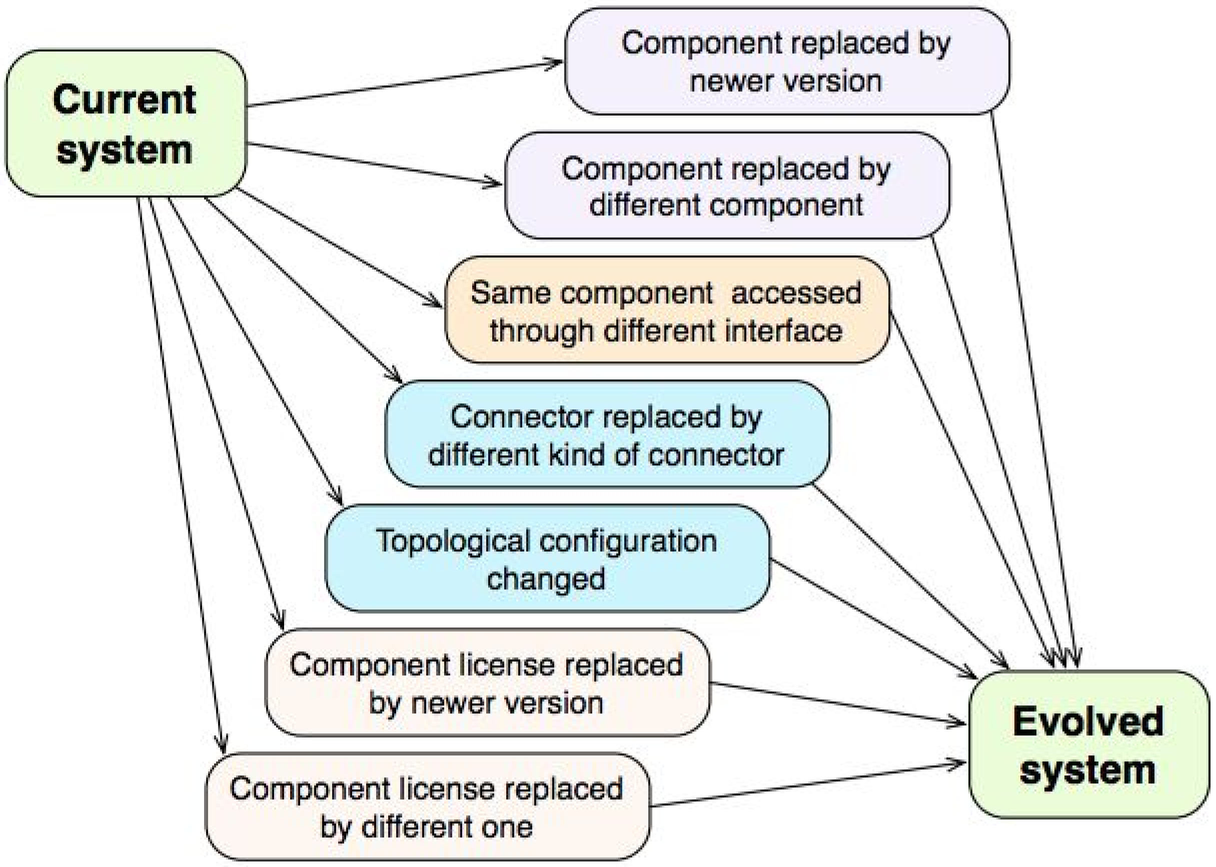
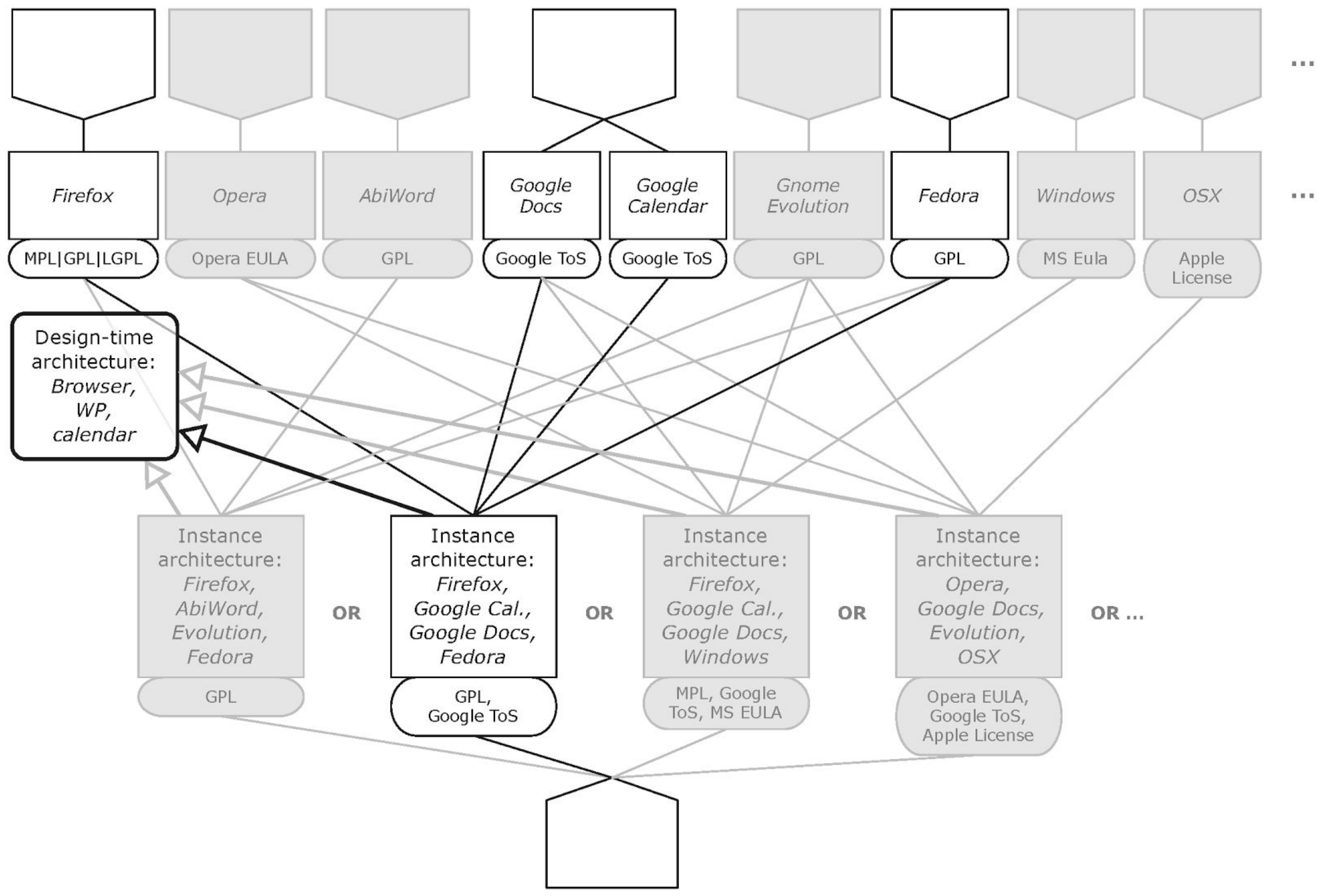

Current system
Component replaced by newer version
Component replaced by different component
Same component accessed through different interface
Connector replaced by different kind of connector
Topological configuration changed
Component license replaced by newer version
Component license replaced by different one
Evolved system

Figure 5: A variety of paths and activities accounting for the evolution of OA systems [ScA12].

Component version evolution— One or more components can evolve, altering the overall system's characteristics. An example is upgrading the Firefox Web browser from version 17.0 to 17.1. Such minor versions changes generally have no effect on system architecture. However, many large enterprises choose to sustain their software systems by relying on "long-term support" (LTS) versions of software components, rather than automatically updating to each release from software component producers. Instead, LTS components are replaced with new versions only over long time frames, where the new LTS version for installation may skip many intervening release versions. Such enterprises rely on local patches and workarounds between the LTS versions, under the belief that such an approach provides increased system stability and allows more comprehensive regression testing prior to deployment. But in these days of relentless attacks on system security, using LTS components entails locally sustaining system component or configuration versions with known vulnerabilities, often without code repositories that match those employed for CI. The vulnerabilities must then be defended using separate, orthogonal system security mechanisms, such as virtual machines or hypervisors from VMWare or Xen, or operating system containers like OpenVZ [Xen13]. Once again, we can do better than this through the use of explicit process specifications that model and provide process integration support across CI and CD systems, along with code repositories.

Component replacement— One or more components may be replaced, each by one or more others with similar functionality and similar interfaces. An example is replacing the AbiWord word processor with either OpenOffice Writer or MS Word, each of which provides roughly the same behavior as a word processor. Other alternative may entail a component with a different user interface plus shim code to make it match its predecessor component, but in different ways. For example, if we replace the AbiWord word processor component, with the Google Docs service, the new word processor's component is now external to the OA system, and in fact could be viewed as now existing within the Web browser component. What these examples reveal is that changes in the format or structure of a component's interconnections, or its APIs, necessitate updates to the build-time and release deployment-time configuration of the component connectors.

Architectural configuration evolution— The OA can evolve by changing the kinds of connectors between components, rearranging connectors in a different configuration, or changing the interface through which a connector accesses a component, altering the system characteristics. Revising or refactoring the configuration in which a component is connected can change how its license affects the rights and obligations for the overall system. An example is the replacement of components for word processing, calendaring, and email with Web-browser-based services such as Google Docs, Google Calendar, and Google Mail. The replacement would eliminate the legacy components and relocate the desired application functionality; it would operate remotely, but interact from within the local Web browser component. The resulting system architecture might be considered simpler and easier to maintain, but is also less open and now subject to a proprietary Terms of Service license. Ongoing evolution and support of this subsystem is now beyond the control and responsibility of the local system developers. System consumer preferences for one kind of license over another, and the consequences of subsequent participation in a different OA system evolution regime, may thus determine whether such an alternative system architecture is desirable or not. Figures 6 and 7 show examples of such evolutions in architectural configuration at release deployment time. These figures can be compared to the system deployment in Figure 4, but now where the build-time architecture now reconfigures the word processor, email and calendaring into the single Web browser component, thus refactoring the build-time and release deployment-time system configurations, while remaining within the design-time product family indicated in Figures 2 and 6.



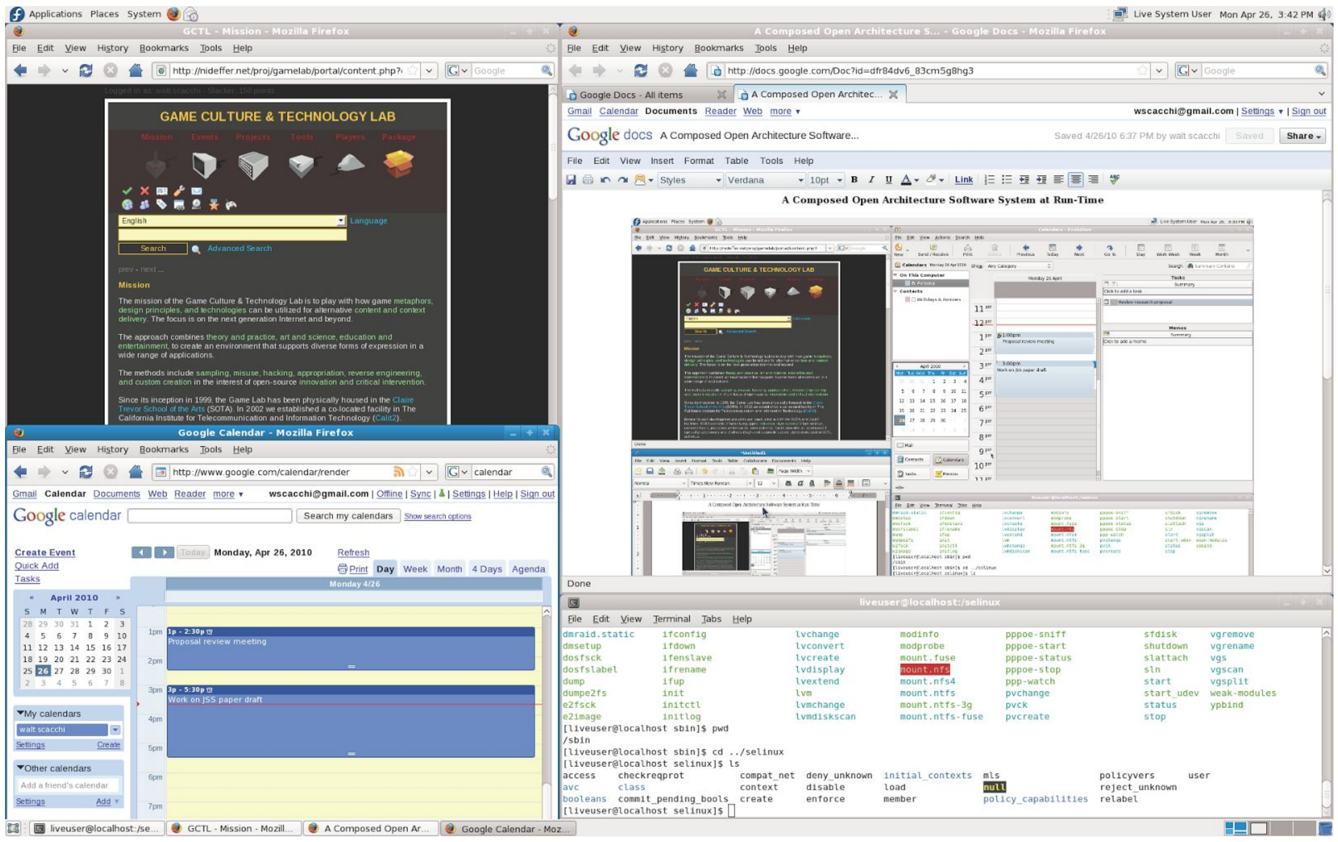
...
Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

Figure 6: An alternative OA system configuration resulting from replacement of selected components shown in Figure 4 during system evolution [ScA12].

Component license evolution—The license under which a component is available may change, as for example when the Mozilla core components changed from dual licensing to the tri-license (MPL, GPL, LGPL). Similarly, when Oracle Corporation took ownership of the Hudson CI system [Kri11], the changes in intellectual property ownership and branding precipitated a major code fork, and instigated parallel independent projects for sustaining development of this OSS CI system [Hud11, Jen13]. Such evolutionary changes, which are common to OSS components, may require reconfiguring an OA system to migrate to a new (re-licensed) component version, or to an alternative system configuration [ScA12].

In response to different 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 in response, whether by changing components, evolving the architecture, or other means, to provide the desired rights within the scope of the acceptable obligations.

Rapid dynamic system reconfiguration— More advanced evolution scenarios entail support for building and releasing of multi-variant system deployment configurations that substitute functionally equivalent software component compilations that produce multiple, diverse executable binary images, each of which may execute in its own processor core, in a multi-threaded, multi-core processor [JSH11]. Pursuing this new path requires a new compilation and build system regime, that in turn anticipates a new generation of CI and CD systems as future research subjects. As should be clear, our purpose is not to provide a prescriptive model of the OA system evolution process, but instead to illuminate how different OA system evolution paths and activities point to issues in process specification, process integration, and the integration of different process enactment representations and mechanisms that must span/link manual-to-automated process hand-offs.



GAME CULTURE & TECHNOLOGY LAB
Google docs
A Composed Open Architecture Software System at Run-Time
Google Calendar - Mozilla Firefox
Google calendar
Monday, Apr 26, 2010
1p - 2:30p Proposal review meeting
3p - 5:30p Work on JSS paper draft
liveuser@localhost:/selinux

Mission

The mission of the Game Culture & Technology Lab is to play with how game metaphors, design principles, and technologies can be utilized for alternative content and context delivery. The focus is on the next generation Internet and beyond.

The approach combines theory and practice, art and science, education and entertainment, to create an environment that supports diverse forms of expression in a wide range of applications.

The methods include sampling, misuse, hacking, appropriation, reverse engineering, and custom creation in the interest of open-source innovation and critical intervention.

Since its inception in 1999, the Game Lab has been physically housed in the Claire Trevor School of the Arts (SOTA). In 2002 we established a co-located facility in The California Institute for Telecommunication and Information Technology (Calit2).

```
dmraid.static     ifconfig          lvchange      modinfo           pppoe-sniff      sfdisk      vgremove
dmsetup           ifdown            lvconvert     modprobe          pppoe-start      shutdown    vgrename
dosfsck           ifenslave         lvcreate      mount.fuse        pppoe-status     slattach    vgs
dosfslabel        ifrename          lvdisplay     mount.nfs         pppoe-stop       sln         vgscan
dump              ifup              lvextend      mount.nfs4        ppp-watch        start       vgsplit
dumpe2fs          init              lvm           mount.ntfs        pvchange         start_udev  weak-modules
e2fsck            initctl           lvmchange     mount.ntfs-3g     pvck             status      ypbind
e2image           initlog           lvmdiskscan   mount.ntfs-fuse   pvcreate         stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot         compat_net  deny_unknown  initial_contexts  mls                   policyvers      user
avc       class                context     disable       load              null                  reject_unknown
booleans  commit_pending_bools create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$ 
```

Figure 7: A screenshot view of a deployed release configuration of the alternative OA system configuration resulting from system evolution [ScA12].

OVERALL OA DEVELOPMENT AND EVOLUTION PROCESS ISSUES

Following from the software processes we examined in our case study and our review of related efforts, we see a number of issues for new software process research emerging. At least six such issues can be identified as follows.

First, we find that a central goal of process automation with widely available software integration and release deployment tools is to find enactment errors and articulation problems more quickly, rather than to provide prescriptive process guidance. Such process enactment details cannot be easily anticipated in general, so process specification and enactment must rely on trial and error, as well as process discovery [JeS06] to surface where additional/new process knowledge is to be found. Consequently, it is not surprising to observe the rise of a new class of software developer role, as “buildmeisters”—developers who specialize in addressing the intricacies, quirks, and problems that arise during software integration processes, since such processes remain ad hoc, undefined, and difficult to model or improve.

Second, current continuous software development systems embody process specifications that are opaque, lack generality, and rely on the processing capabilities of specific incorporated tools to structure process enactment actions, decisions, and outcomes. Different CI systems embody different versions or variants of software build, test, and package processes. This implies that merely having a “defined” process model for processes like continuous integration and release deployment means that such a model will either be insufficiently detailed to provide anything beyond introductory level guidance, or more completely detailed but idiosyncratic because it is bound to specific process automation tools. This in turn makes the process specification problematic to adapt and evolve. There is a basic need for richer process models that represent both the idiosyncratic details of process automation tools for continuous integration and release management, and the generalized abstractions of such processes that can be reused for process (design) guidance and tailoring in specific software development organization settings (cf. [RKA11]).

Third, a recurring challenge from a process research standpoint is how to specify, model, analyze, or simulate software processes that span from mostly manual to mostly automated process enactment activities.

Fourth, automated process enactment systems are themselves subject to continuous improvement and evolution. This means the processes being supported are potentially evolving. However, if their process specification or model is tacit, or is encoded in implementation details, then the process may be opaque to all except the tool’s developers. Thus, trying to specify, model, or simulate software processes that employ automated enactment systems, requires the ability to address processes that are co-evolving: i.e., how tool evolution drives development process evolution, and how development process evolution precipitates tool evolution (cf. [Sca06]). So choosing to only attend to one, misses observation or specification of activities that enable or constrain the other. Such a dilemma points to another challenge for new software process research.

Fifth, process guidance specification and enactment automation are easily conflated in continuous integration and release deployment systems. As a result, developers of OA systems rely on informal best practices to get continuously integrated software products out the door. Separating the specification of such processes from their implementation within the automated system would be an important contribution to the advancement of such systems. Similarly, providing guidance for how to specify processes more abstractly than as low-level process execution script commands (cf. [Hyp13]), would also contribute to the advancement of automated continuous software development systems.

Sixth, the development and evolution of component-based OA systems is both an interesting and a challenging problem for the software process research community. Such systems are likely to follow continuous software processes— processes that are repeatedly enacted hundreds to thousands of times during the sustained life of the system. Such processes are thus appropriate for careful empirical study, simulation, and analysis. The need to address how to continuously secure OA systems further complicates the challenges for software process research. Process streamlining optimizations, opportunities, and guidelines are likely subjects for further research and practical application. Similarly, when the software processes for securing an OA system involve automated process enactment, it appears that compliance testing—checking whether an automated enactment produced a system configuration that is compliant with the system's security policy—will increase in importance. Such compliance is likely to be ad hoc, unless the security policy is formalized into a computational model [ScA13] that can be cross-checked with the enactment results.

Last, empirical study of the software processes of interest, especially as they are observed in different OSS development projects, provides many insights and best practices that can help in the specification (modeling) and integration of processes for developing and evolving secure OA software systems.

CONCLUSION

Process models provide a valuable means for specifying complex software production processes. Such models may have their greatest impact for project and process management, and for coordinating disparate software production processes together with automated enactment tools spread across an ecosystem of software producers. Explicit, open, and sharable process specifications are key to realizing these potential benefits, while the absence of such specifications means lost opportunities to reduce overall software production costs, improve software quality and security, and to streamline and continuously improve such explicit processes.

Managing and coordinating the development and evolution processes for producing secure open architecture software systems is challenging as we have shown in our case study. But as we have observed in our case study, widely available automated technologies for continuous integration and release deployment obscure or hide what these processes are. Further, we find that frequent errors and articulation problems in automated process enactment are expected, since process enactment details are ad hoc and idiosyncratic, while enactment processes are underspecified, not explicit, and encoded in an enactment system's implementation. However, automated process enactment systems may offer the potential to be extended to support (partially) automated process discovery and computational reenactment (cf. [JeS05, JeS06]), rather than just traditional process modeling and simulation. Thus, software producers of contemporary component-based OA systems are working against their self interests, assuming their interests are to improve their productivity and software quality, while reducing avoidable rework and other software production cost drivers.

Our study in this paper sought to identify a range of emerging issues in software process research, especially for process specification/modeling, as well as for process design, automation and integration. Similarly, our case study highlights a number of ways how the need to continually secure an evolving OA system further complicates challenges for software process research. Finally, assuring that software development and evolution processes comply with extant system (or enterprise) security policies—which are presently informal requirements specification documents—means that process compliance checking arises as a practical need unmet by available software process tools.

Overall, our goal in this paper was to employ a case study and related research to help identify and articulate an emerging set of challenges for further software process research and development, Through both a review of related efforts and our case study, we identified a number of challenges for software process research whose investigation and resolution can lead to more streamlined and easier to continuously improve software development and evolution practices that are con- figured for specific organizations, different development tool chains, alternative target system platforms, and secure OA software product families, as well as for their evolutionary reconfiguration.

REFERENCES

[AAS12] T. A. Alspaugh, H. U. Asuncion, and W. Scacchi. Software licenses, open source components, and open architectures. In I. Mistrık, A. Tang, et al., editors, *Aligning Enterprise, System, and Software Architectures*, pages 58–79. IGI Global, 2012.

[AAS13] T. A. Alspaugh, H. U. Asuncion, and W. Scacchi. The challenge of heterogeneously licensed systems in open architecture software ecosystems. In S. Jansen et al., editors, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar Publishing, 2013.

[BrW12] A. Brown and G. Wilson, editors. *The Architecture of Open Source Applications*. Lulu.com, 2012.

[CCL06] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. In *Int. Conf. on Software Eng. Advances* (ICSEA '06), pages 44–54, 2006.

[DIS12] Defense Information Systems Agency. *Network/ Perimeter/Wireless—Wireless* (Smartphone/Tablet), Oct. 2012. http://iase.disa.mil/stigs/net_ perimeter/wireless/smartphone.html.

[Dig12] L. Dignan. Knight Capital future in jeopardy over botched software upgrade. *ZDNet*, 2012. http://www. zdnet.com/knight-capital-future-in-jeopardyover-botched-software-upgrade-7000002116/.

[DMG07] P. Duvall, S. Matyas, and A. Glover. *Continuous integration: Improving software quality and reducing risk*. Addison-Wesley Professional, 2007.

[Fow00] M. Fowler. *Continuous integration* (original version), Sept. 2000. http://martinfowler.com/articles/ originalContinuousIntegration.html .

[GPS94] P. K. Garg, P. Mi, T. Pham, W. Scacchi, and G. Thunquest. The SMART approach for software process engineering. In *16th Int. Conf. on Software Engineering* (ICSE '94), pages 341–350, 1994.

[Giz11] N. Gizzi. Command and Control Rapid Prototyping Continuum (C2RPC) transition: Bridging the valley of death. In *8th Annual Acquisition Research Symposium*, pages 135–154, May 2011.

[GST12] M. M. Gorlick, K. Strasser, and R. N. Taylor. Coast: An architectural style for decentralized on-demand tailored services. In *Joint Working IEEE/IFIP Conf. on Softw. Architecture and European Conf. on Softw. Architecture* (WICSA-ECSA '12), pages 71–80, 2012.

[Gra81] J. Gray. The transaction concept: virtues and limitations. In *7th International Conference on Very Large Data Bases* (VLDB ’81), pages 144–154, 1981.

[Hud11] Hudson-ci. *Hudson best practices*. Eclipsepedia, Aug. 2011. http://wiki.eclipse.org/Hudson-ci/Hudson_ Best_Practices. Accessed 2 Jan 2013.

[HuF10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[Hyp13] Hypertable. *How to build Hypertable on various platforms*, 2013. Accessed 1 Dec 2013. https: //code.google.com/p/hypertable/wiki/HowToBuild .

[IBM07] IBM Software Group. *SW5706 installation wizard hangs*, 2007. Accessed 3 Jan 2013. http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/com.ibm.iea.was_v6/waspdguide/6.0/GettingStarted/Case2_Install_Wizard_Hangs.pdf .

[JSH11] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. In S. Jajodia, A. K. Ghosh, et al., editors, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, pages 77–98. Springer, 2011. http://dx.doi.org/10.1007/978-1-4614-0977-9_4 .

[Jen13] Jenkins. *Upgrading from Hudson to Jenkins*, Jan. 2013. https://wiki.jenkins-ci.org/display/JENKINS/Upgrading+from+Hudson+to+Jenkins .

[JeS05] C. Jensen and W. Scacchi. Process modeling across the Web information infrastructure. *Software Process: Improvement and Practice*, 10(3):255–272, 2005.

[JeS06] C. Jensen and W. Scacchi. Experiences in discovering, modeling, and reenacting open source software development processes. In *Unifying the Software Process Spectrum*, pages 449–462. Springer, 2006.

[Kri11] P. Krill. Oracle hands Hudson to Eclipse, but Jenkins fork seems permanent. *InfoWorld*, May 2011. https: //www.infoworld.com/d/application-development/oracle-hands-hudson-eclipse-jenkins-forkseems-permanent-021 . Accessed 2 Jan 2013.

[MFP06] N. H. Madhavji, J. Fernandez-Ramil, and D. E. Perry, editors. *Software Evolution and Feedback: Theory and Practice*. Wiley, 2006.

[MiS91] P. Mi and W. Scacchi. Modeling articulation work in software engineering processes. In *First Int. Conf. on the Software Process*, pages 188–201, 1991.

[MiS92] P. Mi and W. Scacchi. Process integration in CASE environments. *IEEE Software*, 9(2):45–53, Mar. 1992.

[NaS87] K. Narayanaswamy and W. Scacchi. Maintaining configurations of evolving software systems. *IEEE Trans. on Software Engineering*, 13(3):324–334, 1987.

[RKA11] W. R. Nichols, P. Kirwan, and U. Andelfinger. A manifesto for effective process models. In *2011 International Conference on Software and Systems Process* (ICSSP '11), pages 242–244, 2011.

[QuH08] M. R. J. Qureshi and S. A. Hussain. A reusable software component-based development process model. *Advances in Engineering Software*, 39(2):88–94, 2008.

[Sca06] W. Scacchi. Understanding open source software evolution. In N. H. Madhavji, J. Fernandez-Ramil, and D. E. Perry, editors, *Software Evolution and Feedback: Theory and Practice*, pages 181–206. Wiley, 2006.

[ScA12] W. Scacchi and T. A. Alspaugh. Understanding the role of licenses and evolution in open architecture software ecosystems. *Journal of Systems and Software*, 85(7):1479–1494, July 2012.

[ScA13] W. Scacchi and T. A. Alspaugh. Advances in the acquisition of secure systems based on open architectures. *Cyber Security and Information Systems Journal*, 1(2), 2-16, February 2013.

[SBN12] W. Scacchi, C. Brown, and K. Nies. Exploring the potential of virtual worlds for decentralized command and control. In *17th Int. Command and Control Research and Technology Symposium* (ICCRTS), 2012.

[Sma12] S. Smalley. The case for Security Enhanced (SE) Android. *2012 Android Builder's Summit*, Feb. 2012. https://events.linuxfoundation.org/images/ stories/pdf/lf_abs12_smalley.pdf .

[Smi11] P. Smith. *Software Build Systems: Principles and Experience*. Addison-Wesley Professional, 2011.

[TCI12] *Thoughtworks CI feature matrix*, 2012. Accessed 2 Jan 2013. http://confluence.public.thoughtworks. org/display/CC/CI+Feature+Matrix .

[USC11] US-CERT. *Architecture and Design Considerations for Secure Software Development. Software Assurance Pocket Guide Series*. U.S. Dept. of Homeland Security, 2011. https://buildsecurityin.us-cert.gov/ swa/downloads/Architecture_and_Design_Pocket_ Guide_v1.3.pdf .

[Wik12] Wikipedia. *Continuous integration*, Dec. 2012. http: //en.wikipedia.org/wiki/Continuous_integration .

[Win13] *WineHQ*. http://winehq.org . Accessed 10 Jan 2013.

[Xen13] *Xen hypervisor project*. http://xen.org/products/xenhyp.html and *OpenVZ*. http://openvz.org Accessed 10 Jan 2013.

Chapter 8.

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

Chapter 8.

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

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 system 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 piece-meal with little/no support for guiding the what system security requirements must address across different system processing elements and data levels, and how those can be manifest 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 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 [Sm12, STIG11] or other IT systems relying on open source technologies [DoD10, Ga10, Gi11, Navy10]. In particular, such study may be of value to securing new cyber warfare technologies [DoD11, SBN11]. 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 [Stux11]. 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 [H11]), and social engineering techniques [Saw11]. 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 systems security mechanisms for enabling security requirements or policies are often employed on an *ad hoc* basis, since there are not 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 [LSM98, SSL99], such as:

- mandatory access control lists;
- firewalls;
- multi-level security;
- authentication (including certificate authority and passwords);
- cryptographic support (including public key certificates);
- encapsulation (including virtualization, hidden vs. public APIs), hardware confinement (memory, storage, and external device (port) isolation) [SWZ12], and type enforcement capabilities;
- secure programming practices (including secure coding standards, data type and value range checking) [Se08];
- data content or control signal flow logging/auditing;
- honey-pots and traps;

- security technical information guides for configuring the security parameters for applications [STIG11] and operating systems [Sm12];
- functionally equivalent but diverse multi-variant software executables [Fr10, SJW11].

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 they 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:

- 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; and
- 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 [DoD11]? 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 an 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.
- 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 [Sh11].
- 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” [SA11], 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 a 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. BA05, BA08].

Traditionally, developing robust specifications for non-functional software system security properties in natural language often produces specifications that are ambiguous, misleading, inconsistent across system components, and lacking sufficient details [YC06]. Using a semantic model and logic to formally specify the rights and obligations required for a software system or component to be secure [BA05, BA08, YC06] 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 [AAS09, ASA10, SA08], may therefore be extendable to also address OA systems with heterogeneous software security license rights and obligations [SA11, AIS12]. Furthermore, the idea of common or reusable software security licenses may be analogous to the reusable security requirements templates proposed by Firesmith [F04] 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 [AIS12] 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 a 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. 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 software (OSS) licenses, for which fulfilling some of the obligations requires parts of the rights that are granted. Also particularly for OSS 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.

In general terms, a security license is analogous to a software copyright license such as GPL (GNU General Public License) [GPL07]. 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 [SA11] has identified rights and obligations such as:

- 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
- 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 [AAS09, AAS11, ASA10]. 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 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 architecture-mediated, 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 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 [SWZ12]: For example, we can find *functionally similar* alternatives from software (component) producers of web browsers like Mozilla (Firefox, Camino, Sea Monkey) vs. Google (Chrome) vs. Microsoft (Internet Explorer), vs. others. Similarly, for word processors, we find alternatives including Microsoft (Word) vs. abisoft.com (AbiWord) vs. Google (Google Docs, which is a remote Web service rather than a component), vs. 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.

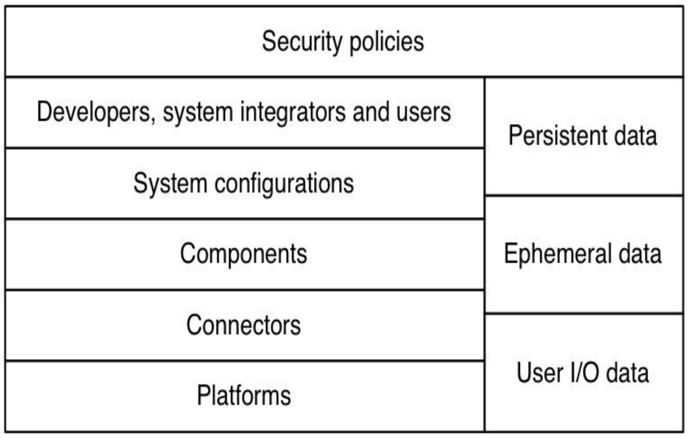
Functionally similar components and services may not be exactly interchangeable, unless their interfaces are similar or identical. As such, it may be necessary to modify, for example, OA system topology, replace connector types, and other architectural measures may be necessary to change from one producer 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* [NS86, Bo06] or a *product line* [CN01].

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 [Fr10, SJWWF11]. Last, software product lines can be bound to a network of software producers, system integrators, and system users/consumers through a software ecosystem [Bo09], such that secure systems can be realized through composition or configuration at the software ecosystem level [SA12]. 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 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.

System security policies provide the overall context for what kinds of security mechanisms or capabilities (e.g., mandatory role-based data access control) are 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.



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

Table 1. Different system security elements whose rights and obligations depend on capabilities supported by lower level elements.

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 [SWZ12]). 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. Operating System (OS), software systems, components, or connectors can each run within their own virtual machine, in in alternative configurations, as long as they are completely confined at a higher level of system security and do not overlap virtual machine boundaries [SSL99, Sm12].
- *network*: message filtering and access control firewalls for data/control flows that move across external hardware system security boundaries.
- *operating systems*: mandatory access control [LSM98, SSL99], capability type enforcement [Sm12], OS configuration parameters [STIG11], 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)
- 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.
- encapsulate 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.
- 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 [Bo06]) enables security by providing:

- ability to host multiple (one or more) alternative, version, or variant system configurations on one or more processors (either single-core [SWZ12], 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.
- 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 [Se08].
- 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 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, user command mouse clicks, 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 of 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.

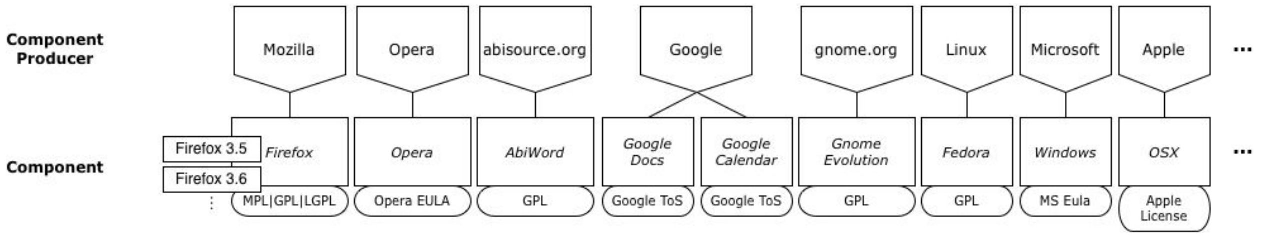


Figure 1. A partial view of a software ecosystem of producers and the software components for an enterprise system they produce

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?

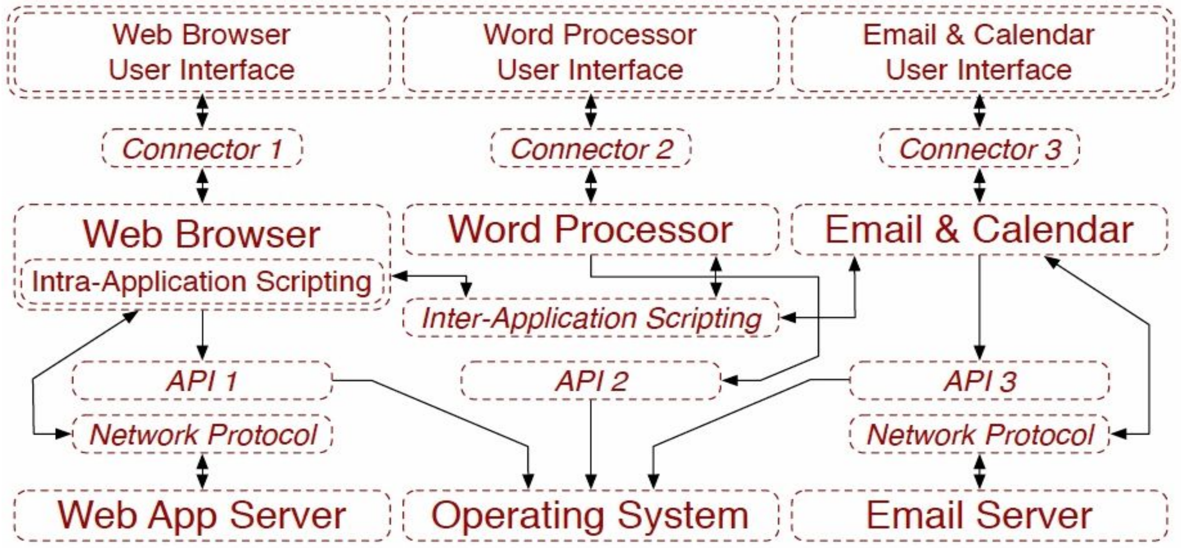


Figure 2. A design-time reference model of an OA system that accommodates multiple alternative system configurations.

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 [AAS09, ASA10, SA11].

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, or by requiring that the network protocol connector be HTTPS.

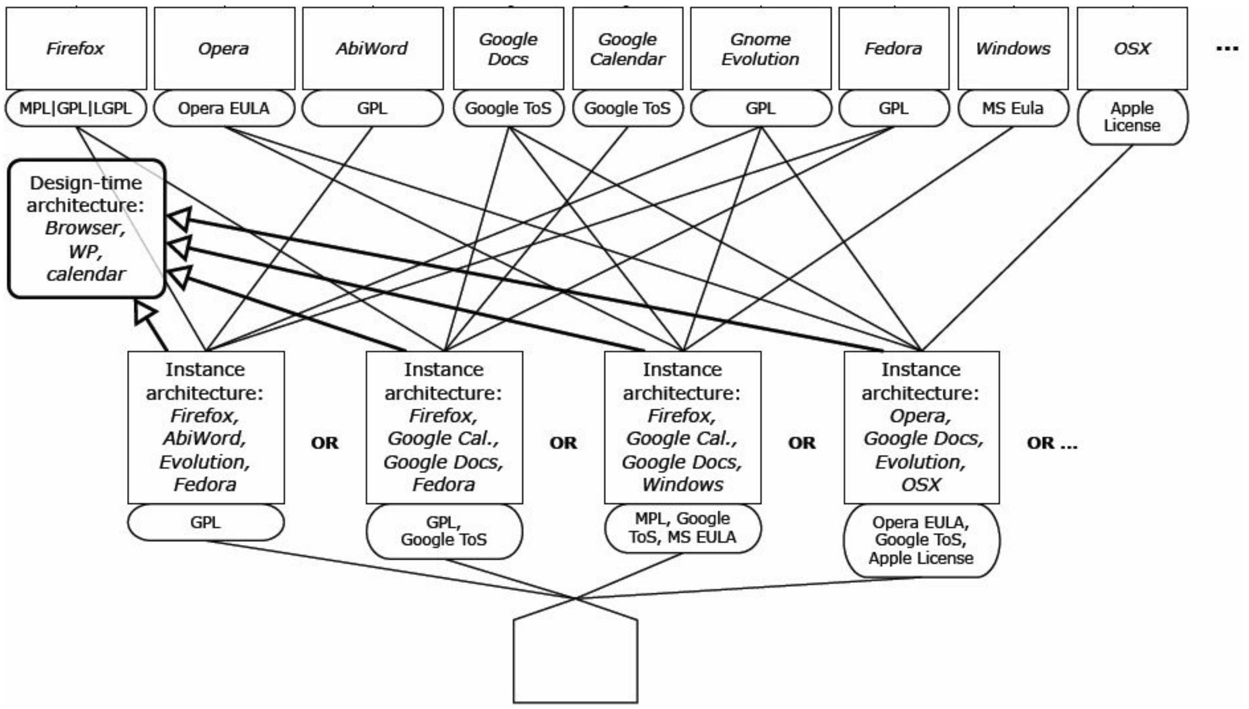
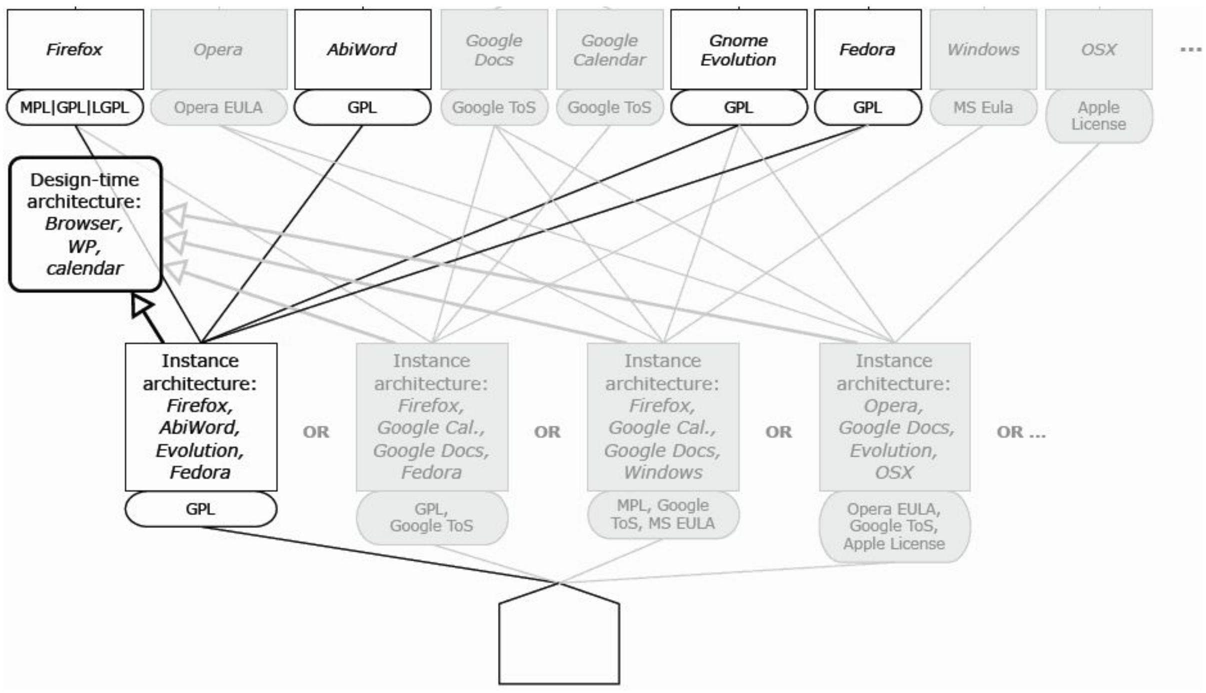

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
OR ...
GPL
GPL, Google ToS
MPL, Google ToS, MS EULA
Opera EULA, Google ToS, Apple License

Figure 3. A view of an OA software ecosystem that provides alternative, functionally similar components compatible with the reference design-time architecture.

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).

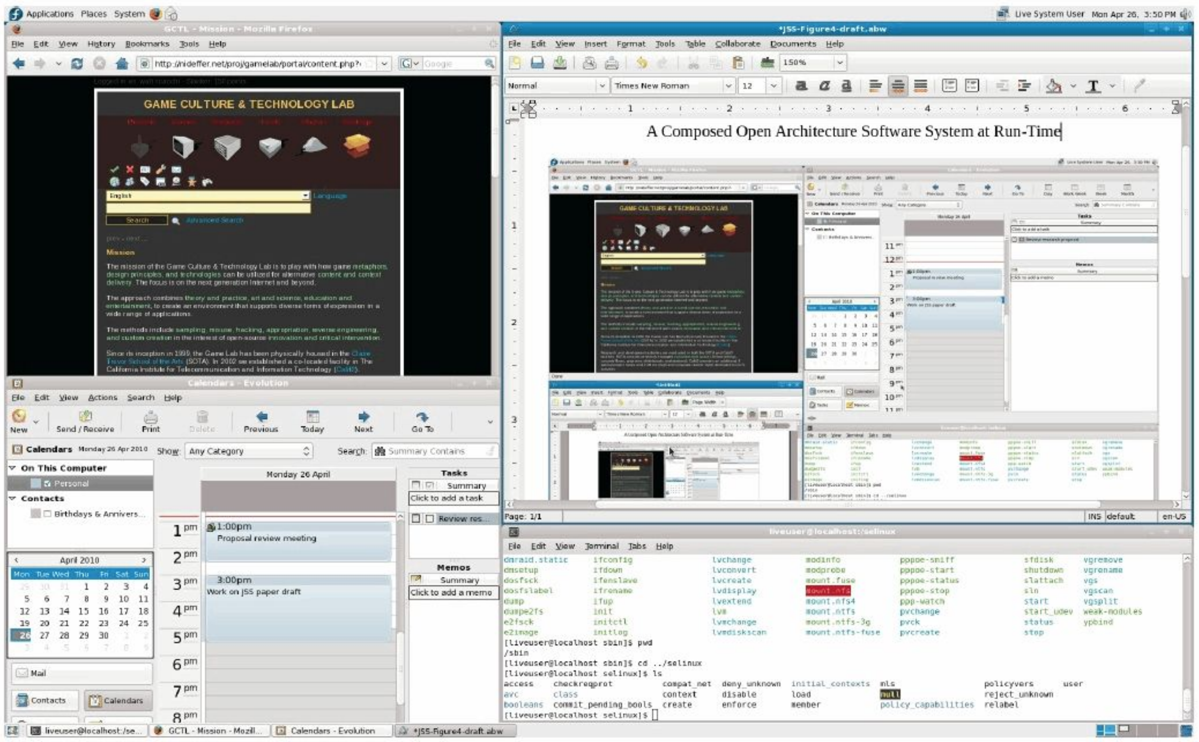
Acquisition-time requirements such as the use of SE Linux and the use of HTTPS could be satisfied by this choice; with an appropriate architecture, the IP licensing obligations could also be satisfied. At design time the functional requirements would need to be satisfied by appropriately-specified shims 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 type of connector that does not transmit IP obligations or rights, though allows for control flow integration, and data flow interoperation.



Firefox
MPL|GPL|LGPL
Opera
Opera EULA
AbiWord
GPL
Google Docs
Google ToS
Google Calendar
Google ToS
Gnome Evolution
GPL
Fedora
GPL
Windows
MS Eula
OSX
Apple License
...
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

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.



A Composed Open Architecture Software System at Run-Time

```
dmraid.static     ifconfig      lvchange     modinfo          pppoe-sniff     sfdisk      vgremove
dmsetup           ifdown        lvconvert    modprobe         pppoe-start     shutdown    vgrename
dosfsck           ifenslave     lvcreate     mount.fuse       pppoe-status    slattach    vgs
dosfslabel        ifrename      lvdisplay    mount.nfs        pppoe-stop      sln         vgscan
dump              ifup          lvextend     mount.nfs4       ppp-watch       start       vgsplit
dumpe2fs          init          lvm          mount.ntfs       pvchange        start_udev  weak-modules
e2fsck            initctl       lvmchange    mount.ntfs-3g    pvck            status      ypbind
e2image           initlog       lvmdiskscan  mount.ntfs-fuse  pvcreate        stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot          compat_net  deny_unknown  initial_contexts  mls                   policyvers      user
avc       class                 context     disable       load              null                  reject_unknown
booleans  commit_pending_bools  create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$ 
```

Figure 5. An end-user run-time version of the selected alternative components that fulfills the design, where the SE Linux operating system (lower right corner) can utilize security modules library for coding and enforcing mandatory access control on programs/data, and other security capabilities.

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 many ways to satisfy the IP constraints imposed by 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 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).

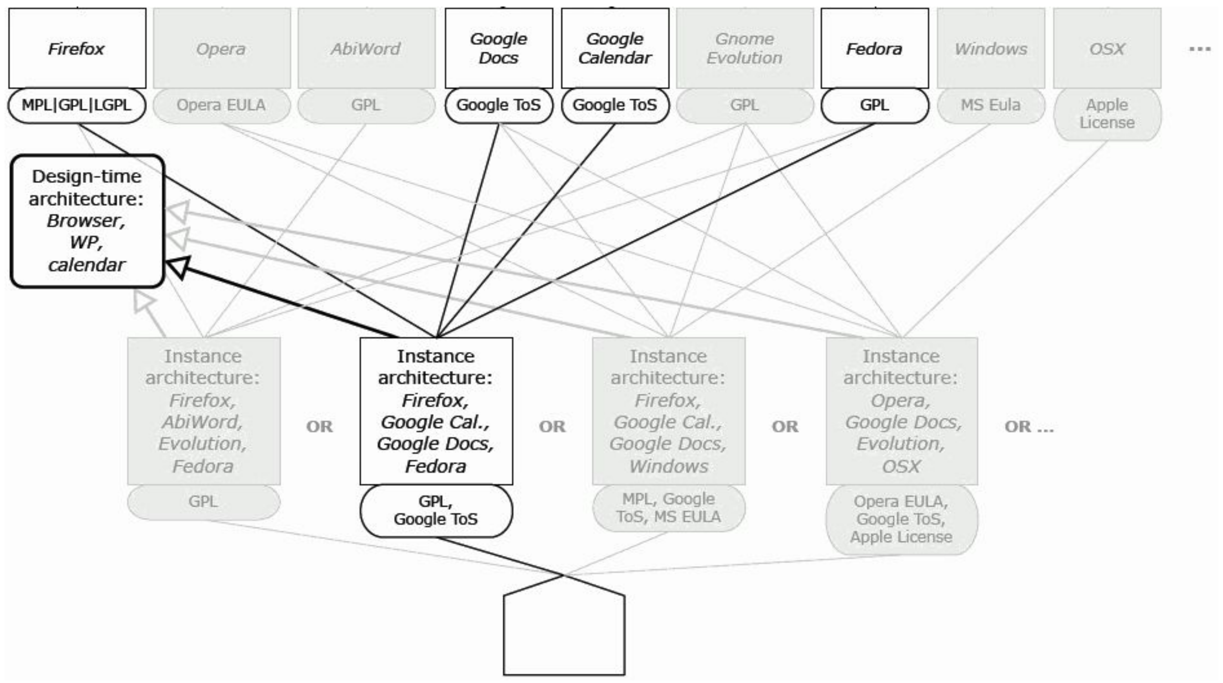
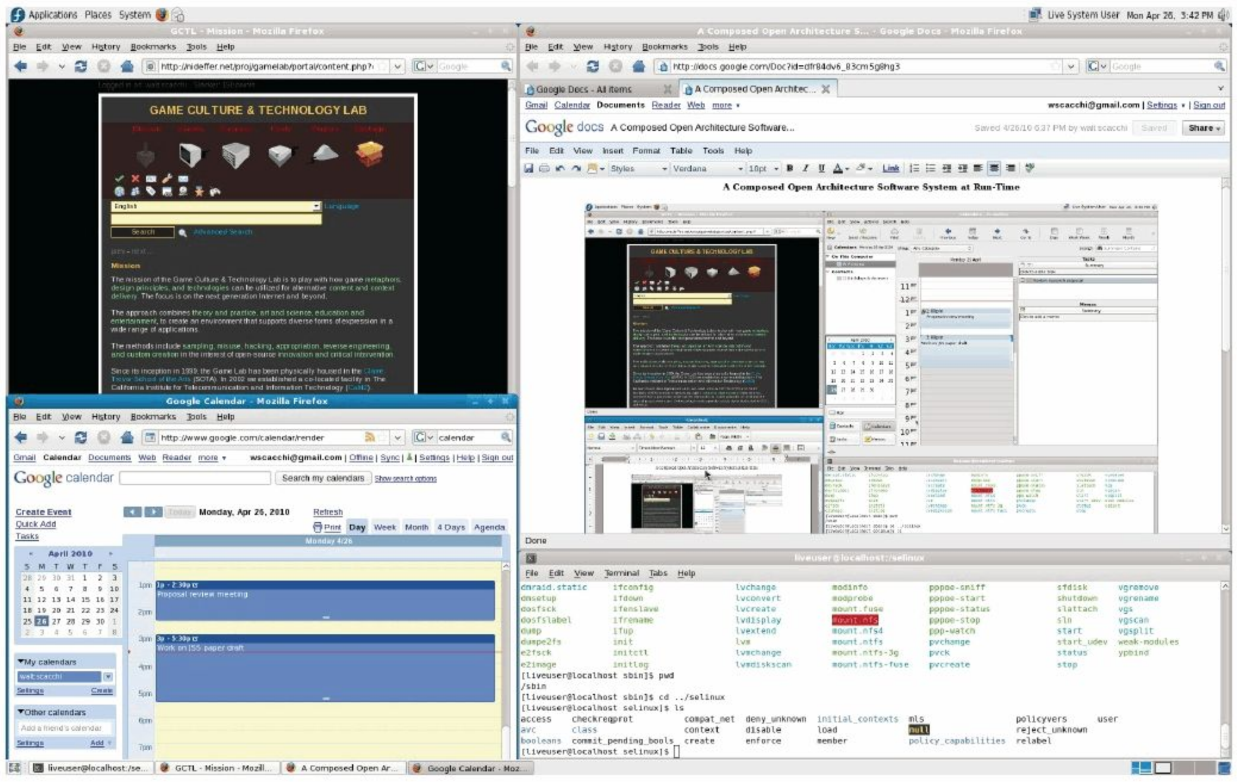

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

Figure 6. An second system configuration, using alternative but functionally similar components.

- 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 we will see below, 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.

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?

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.



```
dmraid.static   ifconfig    lvchange    modinfo         pppoe-sniff   sfdisk      vgremove
dmsetup         ifdown      lvconvert   modprobe        pppoe-start   shutdown    vgrename
dosfsck         ifenslave   lvcreate    mount.fuse      pppoe-status  slattach    vgs
dosfslabel      ifrename    lvdisplay   mount.nfs       pppoe-stop    sln         vgscan
dump            ifup        lvextend    mount.nfs4      ppp-watch     start       vgsplit
dumpe2fs        init        lvm         mount.ntfs      pvchange      start_udev  weak-modules
e2fsck          initctl     lvmchange   mount.ntfs-3g   pvck          status      ypbind
e2image         initlog     lvmdiskscan mount.ntfs-fuse pvcreate      stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot          compat_net  deny_unknown  initial_contexts  mls                   policyvers     user
avc       class                 context     disable       load              null                  reject_unknown
booleans  commit_pending_bools  create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$ 
```

Figure 7. An end-user view of the alternative run-time system 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.

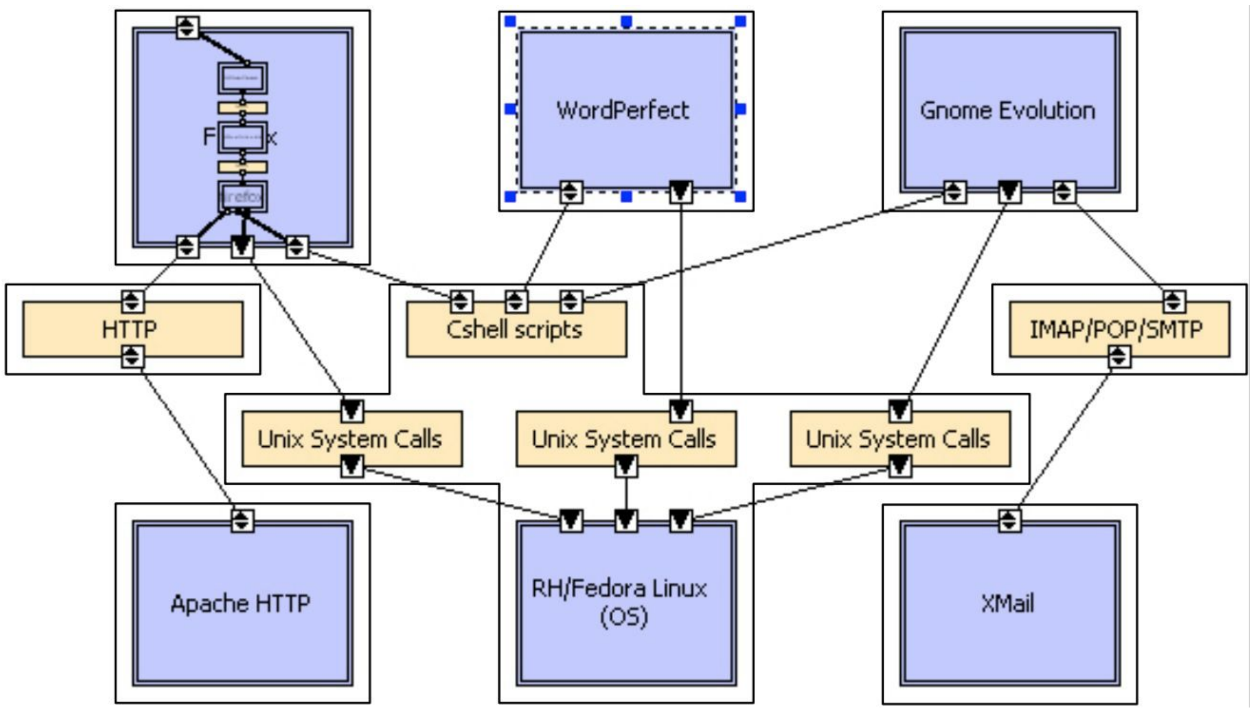


Figure 8. A security configuration alternative for the run-time configuration instance that encapsulates OA system components and connectors within different security containers (e.g., using virtual machines [Xen12] or virtual operating system instances via OpenVZ).

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 within one (for minimal security confinement) or more (for increased security confinement on a per component basis) Xen or OpenVZ-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 500K Linux systems running Xen) [Prg12, Xen12].

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 [SA08, SA11].

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 using the kind of system security licenses we anticipate [SA11], 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. and Scacchi, W. (2009). Intellectual Property Rights Requirements for Heterogeneously Licensed Systems. In *Proc. 17th IEEE International Requirements Engineering Conference (RE'09)*, 24–33, Aug. 31–Sept. 4 2009.

[AAS11] Alspaugh, T.A., Asuncion, H. and Scacchi, W. (2011). Presenting Software License Conflicts through Argumentation, *Proc. 23rd. Intern. Conf. Software Engineering and Knowledge Engineering*, July 2011.

[AlS12] Alspaugh, T.A., and Scacchi, W. (2012). Licensing Security, *Proc. Fifth Intern. Workshop on Requirements Engineering and Law*, 25-28, September 2012.

[ASA10] Alspaugh, T.A., Scacchi, W., and Asuncion, H. (2010). Software Licenses in Context: The Challenge of Heterogeneously Licensed Systems, *J. Association for Information Systems*, 11(11), 730-755, November 2010.

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

[Bo09] Bosch, J., (2009). From software product lines to software ecosystems. In: *Proc. 13th Intern. Software Product Line Conference* (SPLC'09), 111-119.

[BA05] Breaux, T.D. and Anton, A.I. (2005). Analyzing goal semantics for rights, permissions, and obligations. In *Proc. 13th IEEE International Conference on Requirements Engineering (RE'05)*, 177–188.

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

[CN01] Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns. Addison-Wesley, New York.*

[DoD10] DoD Open Source Software (OSS) FAQ (2010). *Frequently Asked Question regarding Open Source Software (OSS) and the Department of Defense (DoD)*. http://cio-nii.defense.gov/sites/oss/Open_Source_Software_%28OSS%29_FAQ.htm

[DoD11] *Department of Defense Strategy for Operating in Cyberspace*, July 2011. http://www.defense.gov/news/d20110714cyber.pdf

[F04] Firesmith, D. Specifying reusable security requirements. *Journal of Object Technology*, 3(1), 61-75, Jan-Feb. 2004.

[Fr10] Franz, M. (2010). E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism, *New Security Paradigms Workshop (NSPW'10)*, Sept. 21–23, Concord, Massachusetts, USA.

[Ga10] Garcia, P. (2010). Maritime C2 Strategy: An Innovative Approach to System Transformation, *Proceedings 15th. International Command & Control Research & Technology Symposium*, Paper 147, Santa Monica, CA.

[Gi11] Gizzi, N. (2011). Command and Control Rapid Prototyping Continuum (C2RPC) Transition: Bridging the Valley of Death, *Proceedings 8th. Annual Acquisition Research Symposium*, Vol. 1, Naval Postgraduate School, Monterey, CA.

[GPL07] *GNU General Public License*. http://www.gnu.org/licenses/gpl.html

[H11] Attack of the Computer Mouse, *The H Online Security*, 29 June 2011. http://h-online.com/-1270018

[LSM98] Loscocco, P., Smalley, S., Muckelbauer, P., Taylor, R., Turner, S. and Farrell, J. (1998). The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environment. *Proc. 21st National Information Systems Security Conference*, 303-314.

[Navy10] Navy.mil (2010). PEO IWS Releases Open Architecture Contract Guidebook Update, http://www.navy.mil/search/display.asp?story_id=53661. Also see *Navy Open Architecture Guidelines* and related documents at https://acc.dau.mil/oa.

[NS87] Narayanaswamy, K. and Scacchi, W. (1987) Maintaining Configurations of Evolving Software Systems, *IEEE Trans. Software Engineering*, 13(4), 323-334.

[Prg12] *SELinux on Xen* (2012). http://wiki.prgmr.com/mediawiki/index.php/SELinux_on_Xen

[SJW11] Salamat, B., Jackson, T., Wagner, G., Wimmer, C., Franz, M. (2011). Run-Time Defense against Code Injection Attacks using Replicated Execution, *IEEE Transactions on Dependable and Secure Computing, Volume 8, No. 4*, July 2011.

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

[SA08] Scacchi, W. and Alspaugh, T. (2008). Emerging Issues in the Acquisition of Open Source Software within the U.S. Department of Defense, *Proc. 5th Annual Acquisition Research Symposium*, Vol. 1, 230-244, NPS-AM-08-036, Naval Postgraduate School, Monterey, CA.

[SA11] Scacchi, W. and Alspaugh, T. (2011). Advances in the Acquisition of Secure Systems Based on Open Architectures, in *Proc. 8th. Annual Acquisition Research Symposium*, Monterey, CA, May 2011.

[SBN11] Scacchi, W., Brown, C., and Nies, K. (2011). *Investigating the Use of Computer Games and Virtual Worlds for Decentralized Command and Control*, Final Report, Grant #N00244-10-1-006, Institute for Software Research University of California, Irvine, July. http://www.ics.uci.edu/~wscacchi/ProjectReports/NPS-Reports/DECENT.pdf

[Se08] Seacord, R. (2008). *The CERT C Secure Coding Standard*, Addison-Wesley, New York.

[Sh11] Shrobe, H. (2011). Secure Computing Systems, Presentation at the *Darpa Colloquium on Future Directions in CyberSecurity*, November, Arlington, VA. www.darpa.mil/WorkArea/DownloadAsset.aspx?id=2147484460

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

[SSL99] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., and Lepreau, J. (1999). The Flask Security Architecture: System Support for Diverse Security Policies, *Proc. Eighth USENIX Security Symposium*, 123-139.

[STIG11] Security Technical Information Guide, *Android 2.2 (Dell).* http://iase.disa.mil/stigs/net_perimeter/wireless/smartphone.html

[Stux11] Stuxnet (2011). Overview at http://en.wikipedia.org/wiki/Stuxnet. Also see M. Falliere, et al., (February 2011), *W32.Stuxnet Dossier*, Version 1.4, http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[SWZ12] Sun, K., Wang,J., Zhang, F. and Stavrou, A. (2012). SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes. *Proc. 19th. Annual Network and Distributed System Security Symposium.*

[Xen12] *Xen Hypervisor Project*, http://www.xen.org/products/xenhyp.html . Also see and compare with *OpenVZ*, https://en.wikipedia.org/wiki/OpenVZ

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

Chapter 9.

Ongoing Software Development without Classical Requirements

Chapter 9.

Ongoing Software Development without Classical Requirements[1]

ABSTRACT

Many prominent open source software (OSS) development projects produce systems without overt requirements artifacts or processes, contrary to expectations resulting from classical software development experience and research, and a growing number of critical software systems are evolved and sustained in this way yet provide quality and rich functional capabilities to users and integrators that accept them without question. We examine data from several OSS projects to investigate this conundrum, and discuss the results of research into OSS outcomes that sheds light on the consequences of this approach to software requirements in terms of risk of development failure and quality of the resulting system.

INTRODUCTION

In 2002 one of us (Scacchi) published a study of requirements practices and artifacts in four open source software (OSS) development communities [Sca02]).This was the first systematic study to show that OSS system and development processes do not rely on what may be termed classical requirements artifacts and processes, namely those involving problem-space requirements in a document or repository evaluated for completeness and internal and external consistency. Others have since reported similar results [Ger03, Nol08, NoL10]. Yet there are successful, ongoing OSS projects with users numbered in the millions, and hundreds of OSS systems relied on as critical infrastructure, such as GNU/Linux, the Apache HTTP server, the Mozilla Firefox Web browser, the PostgreSQL database system, and the Eclipse development platform to name a few [DRW04, MFH02, Pos13, Sta07].

From the point of view of a classically trained software developer and requirements practitioner and researcher such as the other of us (Alspaugh), this is unexpected. The broad consensus among software experts and researchers over recent decades has been that devoting appropriate attention to requirements processes and artifacts is essential to project success [BRo75, GaW89, Jac95, Lam09, Som04, vaV00], and that failure to do so risks undesirable outcomes such as:

- a product that fails to meet stakeholder needs,
- a product that does not exhibit necessary levels of reliability, evolvability, or other software qualities,
- schedule slips and budget overruns, or
- in extreme cases failure to produce any product at all.

[1] This chapter is based on earlier version appearing in *Proc. 21st. IEEE Intern. Conf. Requirements Engineering*, Rio de Janeiro, Brazil, 165-174, 15-19 July 2013.

How can it be that OSS development produces good software?

In the remainder of the paper we explore this conundrum. We first present a motivating example, Brooks's thoughts on the success of Linux, and then elaborate what we mean by "classical requirements artifacts and processes", hereafter abbreviated as Classical Requirements, before describing our study in the next section. We then turn to examining the OSS artifacts and processes that appear to serve in the place of Classical Requirements, using data from our previous work and work reported by others. We find that the overwhelming majority of requirements-like artifacts identified by ourselves and others may be characterized as what we term *provisionments*, which state features or qualities in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies. The processes involving these artifacts resemble or in some cases are indistinguishable from the bug reporting, tracking, and response processes found in closed source software (CSS) development. We discuss several contexts in which provisionments appear common and are arguably appropriate: OSS of course, software game mods, and open architecture software ecosystems.

Finally, we place our work in the context of related work, discuss several questions of interest, and then conclude this chapter.

A MOTIVATING EXAMPLE: BROOKS ON LINUX

In reflecting on Raymond's description Raymond (2001) of the open source process producing Linux, Brooks observes of this "marvelously functional and robust operating system" that "for Linux a functional specification already existed: Unix" (Brooks, 2010, page 56). This is a curious statement, since the development of Unix itself displayed characteristics of OSS development including:

- software developed for the developers' own use rather than for an external client and users,
- a strong emphasis on extensibility, and
- no overt requirements artifacts or process preceding development.

Saying that Unix (specifically the Unix kernel) provided the requirements for Linux does not explain the problem; it merely moves it from Linux to Unix. The Unix kernel is marvelously functional and robust, too; was it developed using a functional specification or other Classical Requirements?

If so, supporting evidence is in short supply. Ken Thompson wrote the initial version of Unix in four weeks in the summer of 1969, yet the first edition of the Unix manual was dated 3 November 1971 Salus [Sal94] notes "the only way you could learn [the Unix system] was to sit down with one of the authors and ask questions." Ritchie [Rit84] recalls that in 1969 "Thompson, R. H. Canaday, and Ritchie developed, on blackboards and scribbled notes, the basic design of a file system that was later to become the heart of Unix"; not the requirements, but the design. We have searched the writings of the creators of Unix and researchers reporting on it for Classical Requirements without finding evidence of it.

It appears that it is indeed possible to produce a marvelously functional and robust operating system without the aid of a functional specification or other Classical Requirements.

Brooks goes on to note, as we and others have, that OSS development works because the developers are users, saying “The whole requirements determination is implicit, hence finessed.” He finds no contradiction in ongoing development without Classical Requirements once initial development is successfully complete.

CLASSICAL ARTIFACTS AND PROCESSES

Researchers and practitioners have developed many types of requirements artifacts and many requirements processes. We do not consider any of them in detail here. Instead we focus on three characteristics shared by nearly every such approach with which we are familiar:

1. a requirements document or central requirements repository, defining the system requirements and providing a criterion for whether a particular candidate requirement is or is not a requirement for the system;
2. requirements that are preferentially described in terms of the problem space rather than the solution space; and
3. requirements processes for examining the requirements document/repository for completeness, internal consistency, and external consistency with the domain and stakeholder needs.

These characteristics define what we term in this paper Classical Requirements.

We focus on these characteristics because they figure prominently in many influential requirements approaches and in the requirements practices of working CSS developers we have known or interviewed, and because convincing arguments have been made from them to project success and product quality [Bro76, GaW89, Jac95, Lam09, Som04, vaV00]. Brooks [Bro87] famously says:

The hardest single part of building a software system is deciding precisely what to build. ... No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

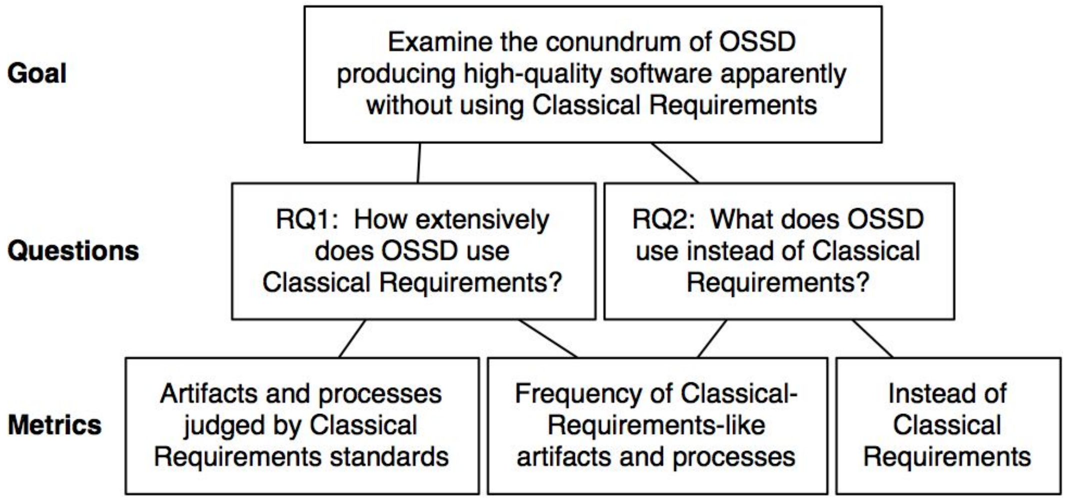
Boehm [Boe76] asserts, supported by data:

Clearly, it pays to invest effort in finding requirements errors early and correcting them in, say, 1 man-hour rather than waiting to find the error during operations and having to spend 100 man-hours correcting it.

Lamsweerde [Lam09] characterizes requirements errors as “numerous and persistent” and as the most expensive and dangerous of software errors. Gause and Weinberg [GaW89] note “Obviously, requirements are important because if you don’t know what you want, or don’t communicate what you want, you reduce your chances of getting what you want.”

The particular form of the requirements is not material to our work. We note that the prominence and importance of particular requirements artifacts and processes often vary depending on the type of system. Not all are appropriate for development of every system, but many situations can benefit from an appropriately chosen selection of them. Some (overlapping) types and corresponding artifact or process choices might be:

- Embedded systems, in which software is a component of a larger hardware system — a state-based specification;
- Real-time systems that must meet specific often-inflexible timing constraints — a temporal logic specification;
- Critical or high-assurance systems, for which what is required and what is acceptable must be determined with precision and the cost of failure is high — a model-checkable specification and validation by stakeholders;
- Systems that interact significantly with other automated systems — a formalized specification checked for consistency and completeness;
- Systems that play a role in specific organizational processes — stakeholder analysis;
- Systems that address novel problems or address problems in a novel way — processes that encourage exploration of the problem space.


Goal
Examine the conundrum of OSSD producing high-quality software apparently without using Classical Requirements
Questions
RQ1: How extensively does OSSD use Classical Requirements?
RQ2: What does OSSD use instead of Classical Requirements?
Metrics
Artifacts and processes judged by Classical Requirements standards
Frequency of Classical-Requirements-like artifacts and processes
Instead of Classical Requirements
Figure 1. Goal Question Metric model

We note that the use of Classical Requirements in these situations and others may be connected to the typical CSS context in which

- the system is produced by a development group for a client outside that group,
- most or all of the system's expected users are also outside that group,
- the developers may or may not have expertise in the problem domain, and
- the system is developed against a budget and a schedule.

The requirements state the expectations and commitments of the client on the one hand and the development group on the other. The client balances the benefits of the specific proposed system against the cost of developing it and the wait until it is ready. The development group evaluates whether the budget, resources, and schedule are appropriate for the work involved. The two sides explore, negotiate, and (ideally) agree on a set of requirements. Both sides can then make plans based on specific criteria for acceptance.

METHOD

Research Questions and Metrics

Our goal is to address the apparent conundrum of OSS development (OSSD) that does not use Classical Requirements yet successfully produces high-quality software. We apply the Goal Question Metric approach [BCr94] to produce a measurement model operationalizing our goal into research questions, and associating each question with data that can be evaluated (Figure 1).

- (RQ1) To what extent do OSS projects in fact use Classical Requirements?
- (RQ2) Where OSS projects do not use Classical Requirements, what artifacts and processes are used instead, if any?

Sources of Data

We address RQ1 and RQ2 using data and results from our previous work [Sca02, Sca09] and from other published research on requirements in OSSD. For an introductory study we find this appropriate, in place of collection of a new set of data. A first step is to identify such research; there is not much. We used work by Noll and Liu [Nol07, Nol08, NoL10] which provides both analysis and some raw data, and work by German [Ger03] providing analysis only. We also examined the data we found while investigating Brooks's statement that Unix provided Linux's function specification, using it primarily to cross-check where possible conclusions we drew from the other data sets. In some cases we followed up on specific data items and examined them in the original context. In a few cases we extended the data with newly-collected data, as for example that shown in Figure 2.

Validity

In this subsection we discuss the internal and external validity of the study, and threats to its validity.

1. *Internal validity*: Internal validity is the soundness of the relationships within a study. Our study examined data and analysis from different researchers, then merged them in order to apply our metrics. We examined original data where possible in order to apply metrics more uniformly. We looked first for overt Classical Requirements, then for requirements-like artifacts and processes, and finally for artifacts and processes that appeared to be used in place of requirements. In order to systematize our study, we coded and categorized each such instance, following standard qualitative practice [Cre03].

2. *External validity*: External validity is the degree to which the results from the study can be generalized. Identification of successful OSS systems without overt Classical Requirements provides an existence proof that software can be successfully developed without it. Other results are more difficult to generalize reliably; for example, the study cannot provide strong support for a hypothesis that Classical Requirements does not contribute to reducing the risk of project failure, nor to increasing the probability that stakeholders will be satisfied. The study also does not provide strong support for hypotheses on the incorporation of OSS development approaches into CSS projects, as our study examines only OSSD data and analyses; these are intriguing and investigation of them remains as future work.
3. *Threats to validity*: We examined every study we found that addressed OSSD requirements, eliminating any possibility of selection bias; however, the number of such studies is quite small (five), making it more difficult to generalize our results and increasing the possibility that other OSSD projects do not fit our conclusions.

Other practitioners and researchers might apply different standards, for example with a broader or stricter definition of which instances qualify as Classical Requirements. We minimized this by defining Classical Requirements explicitly and in abstract terms. This threat affects only RQ1.

OSS ARTIFACTS AND PROCESSES

Requirements-Like Artifacts and Processes

We present several examples of specific requirements-like artifacts and processes we identified in our study. Perhaps the most common requirement-like OSS artifacts are isolated feature requests or bug reports submitted to tracking systems like Bugzilla (Figure 2), and discussed there or on email lists or electronic bulletin boards. An example is this proposal for OpenEMR [NoL10]:

You could add a link to the existing superbill page which would open a new browser window/tab with a printable version that meets your criteria. This way, you could leverage existing code and probably not have to add a table. I am thinking of something similar to printable links elsewhere in the program, like in reports and patient report.

A second example is shown in Figure 2. Here a Firefox feature request is being discussed, in conjunction with possible changes to the implementation and architecture. Comment 4 may be taken as stating a requirement that Firefox provide the Profiler, specifically, and more generally that Firefox provide a specific kind of results (those that the Profiler currently provided, we infer). This fairly explicit requirement is stated in solution-space terms (what Profiler provides) rather than the corresponding problem-space terms; of course, this is probably considerably more compact. The discussion is focused on architecture and implementation issues involved in the requirement. Other requirements are considered only indirectly if at all, for example if the goal of replacing JSD1 with JSD2 + RDP is taken to imply a here-unstated software quality requirement.

A third example is tabbed browsing, a Web browser feature little known not so many years ago, but now so nearly universal that the name “tabbed browsing” has become a token representing a complex of properties and user stories now assumed to be obvious and requiring no explanation.

Mozilla/Firefox tabbed browsing appears to have first been proposed in a one-sentence scenario of use (“One thing that I would really want to see is the ability to open a link in the new window in background . . . ”) posted to a Mozilla newsgroup, which was immediately followed by a post beginning “Have you tried tabbed browsing [in the Opera web browser]?” [Nol07]. Both these are provisionments; the first cites current system behavior and describes a difference from it, while the second cites another system that exhibits the behavior referred to.

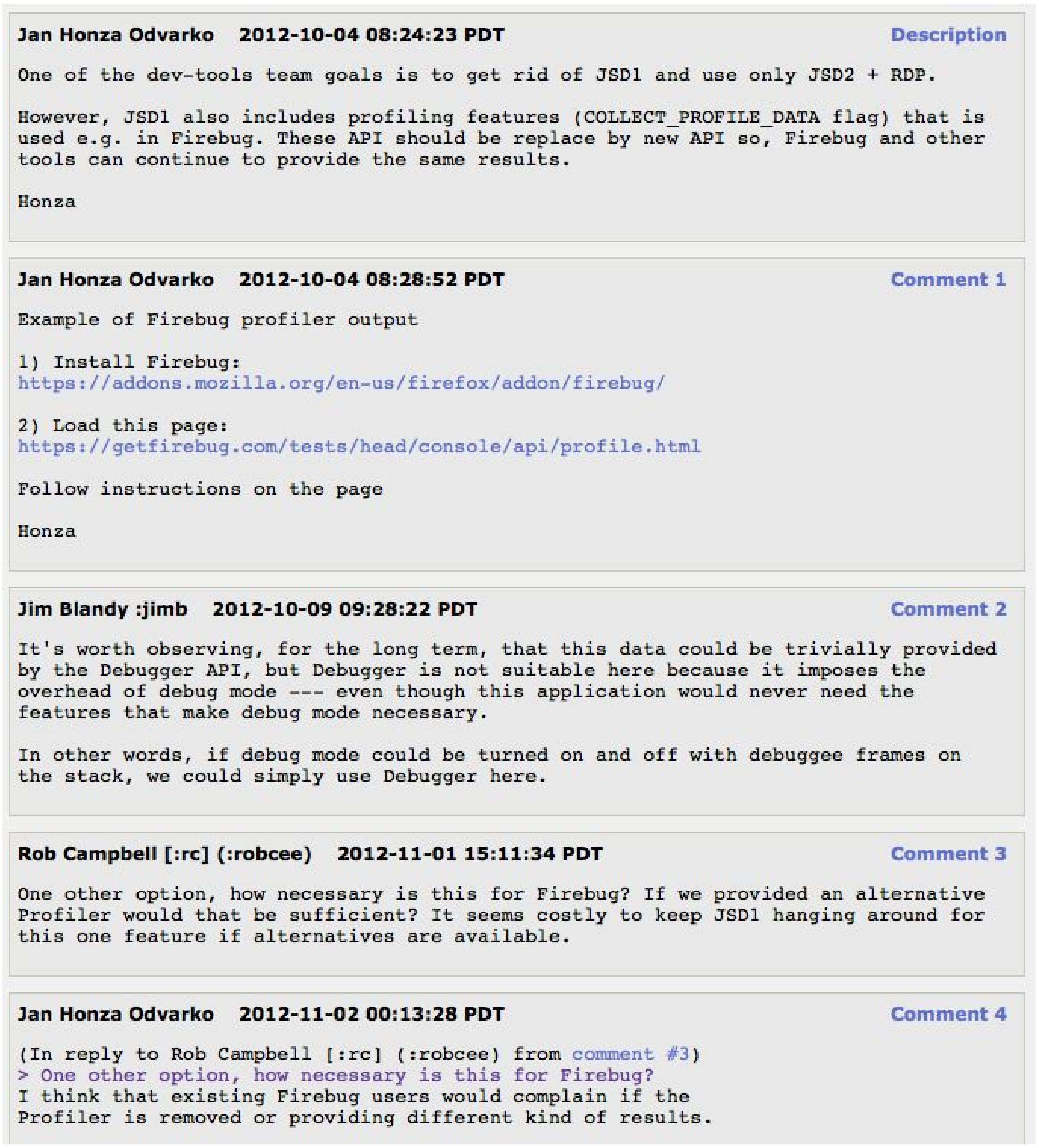
Each feature request or bug report can be taken to imply a requirement, but in themselves they rarely constitute a Classical Requirements artifact. In the examples listed above, as for most requirements-like artifacts we identified, the artifacts are neither integrated into a central requirements document/repository, described in terms of the problem, nor being examined in the context of other requirements. Our study indicates that the OSS projects in question do not use Classical Requirements.

An OSS Requirements Document

Our data sources included one example identified as a requirements document: “Firefox2/Requirements” [MoF06] , discussed by Noll [Nol08]. The document is interesting to us in two ways.

First, our examination of the document found the items are expressed in general rather than specific terms, as in this representative example “[The system] will be optimized and tuned for general web browsing use cases”, with the specifics no doubt proposed, discussed, and agreed on through project mailing lists and discussion boards as in the examples in the previous section. We also note all but one are stated as a difference from the previous Firefox version, using phrases such as “will update” and “will improve”, in other words as provisionments.

Second, and perhaps more significant, this is the only presumptive requirements document or repository our research identified in our own searches and in related work on requirements in OSS. While its existence indicates that OSS development can tend toward Classical Requirements, its apparent uniqueness highlights our general finding that OSS development does not make use of Classical Requirements.


Jan Honza Odvarko 2012-10-04 08:24:23 PDT Description

```
One of the dev-tools team goals is to get rid of JSD1 and use only JSD2 + RDP.

However, JSD1 also includes profiling features (COLLECT_PROFILE_DATA flag) that is
used e.g. in Firebug. These API should be replace by new API so, Firebug and other
tools can continue to provide the same results.

Honza
```

Jan Honza Odvarko 2012-10-04 08:28:52 PDT Comment 1

```
Example of Firebug profiler output

1) Install Firebug:
https://addons.mozilla.org/en-us/firefox/addon/firebug/

2) Load this page:
https://getfirebug.com/tests/head/console/api/profile.html

Follow instructions on the page

Honza
```

Jim Blandy :jimb 2012-10-09 09:28:22 PDT Comment 2

```
It's worth observing, for the long term, that this data could be trivially provided
by the Debugger API, but Debugger is not suitable here because it imposes the
overhead of debug mode --- even though this application would never need the
features that make debug mode necessary.

In other words, if debug mode could be turned on and off with debuggee frames on
the stack, we could simply use Debugger here.
```

Rob Campbell [:rc] (:robcee) 2012-11-01 15:11:34 PDT Comment 3

```
One other option, how necessary is this for Firebug? If we provided an alternative
Profiler would that be sufficient? It seems costly to keep JSD1 hanging around for
this one feature if alternatives are available.
```

Jan Honza Odvarko 2012-11-02 00:13:28 PDT Comment 4

```
(In reply to Rob Campbell [:rc] (:robcee) from comment #3)
> One other option, how necessary is this for Firebug?
I think that existing Firebug users would complain if the
Profiler is removed or providing different kind of results.
```

Figure 2. Discussion of a feature request in the Firefox Bugzilla, “Bug 797876 — Introduce new API for JS content Profiling”

PROVISIONMENTS

As stated in the Introduction, a provisionment is a statement of features or qualities in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies. Most provisionments we encountered only suggest or hint at the behavior or quality in question; the expectation seems to be that the audience for the provisionment is either already familiar with what is intended, or will play with the cited system and see the behavior or quality in question firsthand.

In our study, we saw provisionments being used for requirements or requirements-like artifacts in two ways: either directly as a specification of behavior or quality, or as a starting point in a specification of behavior or quality differing in stated ways from that expressed by the provisionment.

The next section provides examples of both types. Firefox Bugzilla comment 4 in Figure 2 “I think that existing Firebug users would complain if the Profiler is removed or providing [sic] different kind of results” uses a provisionment directly (though stated in the negative), while the OpenEMR proposal uses a provisionment (“the existing superbill page”) indirectly as a starting point for a difference (“You could add ...”).

A provisionment is distinct from a feature, a quality, a bug report, and similar entities in that each of those is something to be expressed, while a provisionment is a way of expressing something. In our study we found many feature requests and bug reports expressed using provisionments; OSS project archives appear to teem with feature requests and bug reports, and the majority we examined were expressed using provisionments. Statements of qualities were much less common but were also often expressed with provisionments.

SOME EXAMPLE CONTEXTS

We discuss three contexts highlighting the interplay between requirements, provisionments, and architecture: open source software, here discussed at greater length; software games, some of which are themselves OSS and many of which support modifications that exhibit OSS characteristics, whether the underlying game is OSS or not, and are described using provisionments; and OA systems of complex components, for which provisionments mediated by architectural configurations play prominent roles.

Open Source Software

OSS requirements, to the extent that they can be identified, tend to be distributed across space, time, people, and the artifacts that interlink them. OSS requirements are thus decentralized—that is, they are decentralized requirements that co-exist and co- evolve within different artifacts, online conversations, and repositories, as well as within the continually emerging interactions and collective actions of OSSD project participants and surrounding project social world. To be clear, decentralized requirements are not the same as the (centralized) requirements for decentralized systems or system development efforts. Traditional software engineering and system development projects assume that their requirements can be elicited, captured, analyzed, and managed as centrally controlled resources (or documentation artifacts) within a centralized administrative authority that adheres to contractual requirements and employs a centralized requirements artifact repository—that is, centralized requirements. In this way as in others, OSSD projects represent an alternative paradigm to that long advocated by software engineering and software requirements engineering community [Sca09].

By the standards of classical software development and requirements practice, OSS requirements and processes are not satisfactory. Requirements are expressed indirectly at best; they are scattered across mailing lists, discussion boards, and bug trackers rather than collected in one place; they appear to be integrated only in the implementation of the system they refer to; they are almost universally stated in solution terms, not problem terms; once stated and discussed, they rarely appear to be referred to.

An RE researcher or practitioner might well look at dispersed statements such as these and simply conclude that requirements were for practical purposes absent by any reasonable or ordinary standard; if such decentralized, indirect requirements were used for a classical software development project, it would be judged to be at high risk of failure.

One would think therefore that many open source projects should fail—and they do, in large numbers. About 59% fail according to one study [WiC10], roughly double the 31% rate at which classical projects are reported to fail according to a 1994 survey [TSG94]. Of course failure means something different for an OSSD project; there is no concept of over budget or behind schedule, and failed OSSD projects tend to wither away rather than being cancelled. Nevertheless, the comparison is startling.

Though most OSSD projects fail to produce a sustained sequence of widely-used software system releases, a substantial number are striking successes. Hundreds of OSSD projects are critical in a number of areas:

- the operation of the World-Wide Web: (the Firefox and Chrome web browsers and the Apache web servers and web services infrastructure);
- interactive software development (Eclipse and NetBeans development environments);
- customer relationship management (SugarCRM);
- database management systems (PostgreSQL, MySQL);
- operating systems (GNU/Linux, Darwin/OSX);
- office communications systems (Asterix), and many more.

Clearly OSSD processes are capable of producing high quality software systems, despite scanty requirements artifacts and processes. We see the use of provisionments to make statements about the functionality of current and future system versions as one key factor, particularly convenient for an ongoing project producing version after version, each of which is described not in absolute terms but in terms of its differences from the previous one. Others may include developing an (informal) architecture and reasoning about it, in place of developing requirements and reasoning about requirements; using extensibility (see below), developer prototypes, and frequent releases of new system versions to explore the problem space by experimenting with alternative solutions within it; the fact that OSS developers are also users of the systems they develop; and the extensive discussions of system issues and proposals, characteristic of OSSD projects, in online forums that are public and persistently available.

We note that many prominent OSS systems are strongly extensible, with mechanisms by which the core functionality of the system may be extended independently, without affecting the system core. These mechanisms allow end-users to customize their copy of a system to suit there own needs and preferences, and in many cases allow developers to expeditiously prototype candidate provisionments. Examples of extensibility include Unix, supporting the addition of shell scripts, commands, libraries, and device drivers; Firefox, Eclipse, jEdit, and others, supporting the addition of plug-ins; and Firefox and jEdit again, and others, supporting the use of scripting languages. In addition to satisfying the system quality requirement (QR) of extensibility, extension mechanisms can also contribute to the requirement, for project success and continuation, to bring new contributors into the project community. Writing extensions for one's one copy of a system is an easy and appealing first step towards making more substantial contributions to the project that produces the system.

Extensibility and several other quality requirements will be seen to play important roles in games and OA systems too.

Viewing OSSD from a classical RE standpoint, we still note some concerns. Classical RE has approaches for identifying relevant stakeholders, and we see no corresponding practice in OSSD. We are concerned that OSSD projects will tend not to identify stakeholder roles in which the stakeholders are not developers and (for whatever reason) not motivated to come forward and contribute. We are also concerned about the effectiveness of OSSD in exploring the problem space, as opposed to the solution space. If such exploration is occurring, it is doing so inconspicuously.

We also do not claim that developers can easily see into their own goals and needs; they are only human, after all. We note only that what corresponds to elicitation may be more straightforward since the communication step vanishes.

Software Game Mods

Many software games are extensible and thus can be modified by their users to produce new games, ranging from simple modifications obviously similar to the host game to others almost unrecognizable as related to their hosts.

User modified computer games, hereafter referred to as game mods, are a leading form of user-led innovation in game design and game play experience. Game mods, modding practices, and modders are in many ways quite similar to their counterparts in the world of OSS development, even though they often seem isolated to those unaware of game software development. Modding is increasingly a part of mainstream technology development culture and practice, and especially so for games. Modders are players of the games they reconfigure, just as OSS developers are users of the systems they develop. There is no systematic distinction between developers and users in these communities, except for the many users/players that contribute little beyond their usage and their demand for more such systems. Modding and OSSD projects are in many ways comparable experiments to prototype alternative visions of what innovative systems might be in the near future, and so both are widely embraced and practiced as a means for learning about new technologies, new system capabilities, new working relationships with potentially unfamiliar teammates from other cultures, and more [Sca07, Sca11].

Game conversion mods are perhaps the most common form of game mods. Most such conversions are partial, in that they add or modify in-game characters, game resources such as weapons, potions, or spells, play levels, zones, landscapes, game rules, or play mechanics. In these cases the conversion can often best be described in terms of provisionments of the host game. More ambitious modders go as far as to accomplish either total conversions that create entirely new games from existing games of a kind that are not easily determined from the originating game, or even parodies that implicitly or explicitly spoof the content or play experience of one or more other games via reproduction and transformation.

One of the most widely distributed and played total game conversions is the Counter-Strike (CS) mod of the Half-Life first-person action game from Valve Software. The CS mod attracted millions of players preferring to play it over the original Half-Life game. Other modders began to further convert the CS mod in part or fully, to the point that Valve Software modified its game development and distribution business model to embrace game modding as part of the game play experience provided by the Half-Life product family. Valve has since marketed a number of CS variants. As of 2011, Valve Software had sold over 25M copies of CS and its descendants [Mak11].

Other player-modders have produced meta-mods, or mods that can themselves be modded, such as Garry's Mod of Half- Life 2. Garry's Mod has evolved into a modding toolkit used in hundreds of game conversions and producing inventive game play mechanics. Game conversions can also exhibit innovations in game design and re-purposing. The game Chex Quest is a conversion of the first-person shooter game Doom into a "non-violent" game distributed in Chex cereal boxes and targeted to young people and gamers (Figure 3).

Extensibility to support the creation of mods has become a necessary feature for a successful game.

Open Architecture Software Ecosystems

As we note in our previous work Alspaugh et al. [AAs09, AAS13, ASA10, ScA12a], a substantial number of development organizations have adopted a strategy in which a software-intensive system is developed with an open architecture (OA) [ORe00, ScA08], integrating components that may be 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 development organization becomes an integrator of components largely produced elsewhere and interconnected through open APIs, with shim code added as necessary to achieve the desired result. This approach allows development of large systems of complex components, with relatively little coding needed. Requirements artifacts and processes are not prominent here. Instead, we see a prototyping process and a system described in terms of provisionments rather than requirements.

CAPTURED P:/MAPPING/MULTIPLAYERDOOM/SKULLTAG/SCREENSHOT_CHEX_20111123_120919.PNG
Chex
LABORATORY
SEYON
ENEMY
RANK: 1/4
SPREAD: 0
62
ZORCH
100%
HEALTH
0
13%
ARMOR
WEAPON

Figure 3. A screenshot of Chex Quest, a nonviolent mod of the Doom game (image courtesy of user *Vulpis Alba*).

One reason that reasoning with provisionments is appealing for OA systems is that the integrator cannot choose arbitrary functional capabilities. Instead, there are a limited number of alternative components to select among, and one must simply take what is available. As the components evolve the same situation recurs, in that the functional capabilities may change from version to version, and the integrator must work with what is available. The most straightforward approach is simply to reason based on what the selected components provide. A second reason is that individual components such as Firefox do not come with Classical Requirements that could be used to reason about requirements for the overall system.

The possible components that can be incorporated into a system define an ecosystem for it. Figure 4 sketches a potential ecosystem for a system composed of a web browser, word processor, email and calendar component, and any scripts and shim code the integrator produces to knit them all together and achieve the desired functionality. If we hypothetically consider the requirements of the composed system, we note that the requirements would necessarily be decentralized, since whatever requirements process we used for the overall system would be independent of that used for each individual component. If we were able to get requirements for each component (which in general is not possible) and integrate them to arrive at requirements for this version of the overall system, this central requirements artifact would last only until the next component version was released, sending the situation back to decentralized requirements.

In practice, integrators appear follow the lead of the developers of the OSS components, and work with provisionments. The acceleration of evolution caused by integrating the independent supply chains for the components currently selected is driving a need to understand decentralized requirements and reason in terms of decentralized provisionments.

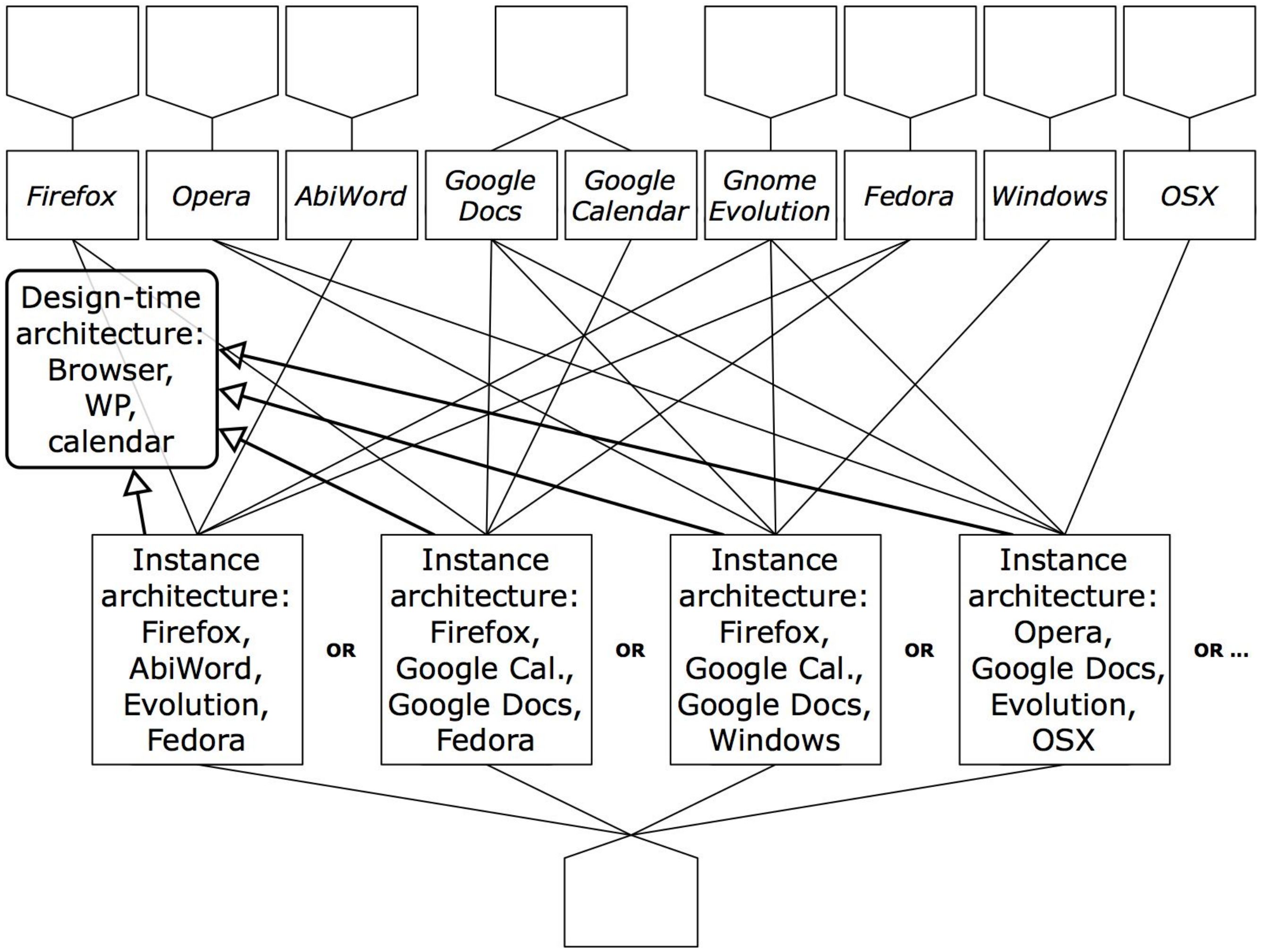

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
OR ...

Figure 4. Ecosystem from which instantiations of the system architecture can be drawn

RELATED WORK

Requirements in open source development

Scacchi was the first to systematically observe and posit the idea that OSS system and development processes do not rely on producing and review of formal functional requirements documents Scacchi (2002). Instead, OSS development projects commonly rely on "software informalisms," no matter what the application domain, nor who the developers may be. Such informalisms are rendered within online artifacts like bug reports, messages in a discussion forum, online chat transcripts, etc. that developers use to communicate their interests about different aspects of a system, its development, its user experience, or its need to evolve in some way. He found that OSS requirements often were described after the functionality they prescribe had already been implemented and found to be viable or practical—requirements after the fact. By 2009 Scacchi [Sca09] had identified a set of twenty-odd different types of informalisms in use across different open source development (OSSD) projects, such that a given project might routinely use a signature set (or genre ecology [Spi03]) of 5-10 informalisms, with different projects utilizing different mixes of software informalisms so that no specific set seems to dominate. The informalisms identified were (a) project email; (b) discussion forums, electronic bulletin boards, and group blogs; (c) news postings; (d) instant messaging; (e) project digests summarizing (a)-(d); (f) usage scenarios as linked Web pages or screenshots; (g) how-to guides; (h) to-do lists; (i) Frequently Asked Questions lists; (j) project Wikis; (k) traditional system documentation; (l) external publications; (m) project licenses; (n) open software architecture diagrams; (o) intra-application functionality in scripting languages; (p) externally developed software modules ("plug-ins"); (q) software modules reused from other OSS projects; (r) project Web sites or portals; (s) project source code Web directories; (t) project repositories such as CVS; (u) bug reports; and (v) issue tracking databases such as Bugzilla. Provisionments may be found in many of these informalisms—especially (a-e), (u), and (v)—but the category of provisionments is orthogonal to them and, we believe, significant in itself.

German [Ger03] described five sources for requirements for the GNOME project, based on his experience as a contributor. He terms them vision (a leader proposes a list of requirements), reference application (an outside system is to be imi- tated), asserted requirement (arising from discussions among contributors), prototype (an implementation illustrating a proposed feature to be discussed), and post-hoc requirement (like a prototype, but offered as a ready-to-integrate implementation of a feature the contributor desires). Provisionments are most closely involved with German's prototypes and post-hoc requirements.

Noll [Nol08] examined the published requirements document for the Web browser version Firefox 2.0, identifying where each of the 14 items was first mentioned, how it was implemented, and why each was initially proposed. Eight were asserted by developers from their personal experience or knowledge of user needs, three were requested by users, and one was driven by a feature in competing browsers. This highlights that although OSS developers are themselves users, non-developer users also play a role in OSS evolution.

Noll and Liu [Nol10] also examined requirements for the OpenEMR electronic medical records project, finding comparable pro- portions contributed by developers vs. users. Each feature was briefly discussed in the project's online developers forum, which they characterized as requirements validation and agreement. We found the OpenEMR requirements or features to be more difficult to classify, for example “Support for deleting immunizations”, and hypothesize that each acts as a token for the corresponding forum discussion.

Requirements and architecture

The close relationship between requirements and architecture suggests that the affordances provided by requirements in classical development may somehow be provided through architectural means in OSSD.

Nuseibeh [Nus01] proposed the Twin Peaks model as an expression of the interrelation of requirements and architecture: problem concerns and solution concerns cannot in general be addressed in sequence, rather needing to be addressed concurrently. The model conveys a back-and-forth alternation treating both requirements and architecture in increasing detail.

De Boer and van Vliet [dVb09] argue that the traditional distinction between requirements and architecture is misguided, and that there is no fundamental difference between them, saying “architecturally significant requirements [ASRs] are in fact architectural design decisions [ADDs], and vice versa”. Both are optative statements characterizing what is desired, and by their nature earlier optative statements constrain what later optative statements can be made.

Alspaugh et al. [DSA07] found that of systems with published development artifacts, only toy systems for textbooks have both complete requirements and a complete architecture. Of the remainder, roughly half had a complete architecture, another quarter had complete requirements, and the remainder had neither. We believe this occurs because requirements and architecture are to a certain degree redundant, so that developers have no need to develop both fully.

All this work suggests that if expected OSS requirements artifacts or processes do not appear to be present, the purposes of those artifacts and processes may be being achieved through architectural means.

DISCUSSION

Are OSS Requirements “Good”?

This is a fascinating question to which we have no definitive answer.

In one sense, the answer is “most definitely not”. The previous career of one of us (Alspaugh) included work as a developer, team lead, manager, and consultant occasionally called in to help struggling development projects. In each case the struggles could usefully be ascribed to problems with the project’s requirements artifacts and processes, in that attacking those problems brought the projects in each case onto a path that could (and usually did) lead to success, and the OSS requirements-like artifacts and processes we examined evoke the problematic ones of those projects.

However, the OSS data we examined in this study was not from troubled projects but from flourishing ones. We conclude that at least some of the work that Classical Requirements accomplishes is being done in another domain with processes appropriate to that domain; our hypothesis, potentially supported by some of the data we examined, is that some of it is being done in the software architecture domain, through processes that are more what we would expect though here again the artifacts do not appear to be overt.

We note again that CSS bug reports and feature requests and the processes for managing them look much like those for OSS.

Centralized vs. decentralized requirements

Rather than a single central requirements or provisionments repository or document, updated as necessary, OSS projects almost universally appear to use email threads, electronic bulletin boards, and similar sequences of archived interactions as a record of them (and of virtually everything else, it appears).

This choice prevents overall consideration and analysis of the provisionments as a whole. However, it may support a deeper goal for OSSD projects: creating and sustaining a community of contributors. The ongoing conversation, archived online so potential contributors can dip into it to see if interests them, provides an ongoing sequence of nudges to participate and a continuing reinforcement of community membership to those who do participate. This may more valuable and fundamental than any incremental benefits likely to accrue from unifying the information into a single document.

Is OSSD efficient?

There does not appear to be data on this question yet. It is not clear that successful OSS projects produce results as expeditiously or more so than CSS projects do; they may well be slower in calendar time or take more person-months. Certainly the importance of schedules and budgets in CSS could drive more efficient development. Brooks [Bro10] notes that one would expect communication to be a more serious bottleneck for OSS than for CSS, though we note this may be ameliorated by the reduction or elimination of communication between developers and stakeholders, since OSS developers are themselves users and stakeholders.

Would OSS Benefit from classical requirements engineering?

Perhaps, but the answer is not clear; at this stage, we can only speculate. If the user-developers are identifying stakeholder needs sufficiently well and those needs are addressed sufficiently well by the incremental revisions that appear to characterize OSSD, then probably not. However if the needs would be best addressed by a reconsideration of the problem and a more radical change in the solution, Classical Requirements has advantages to offer.

We note the truism that a new solution to a problem opens the eyes of its users to new problems not previously considered. A product that is evolving at a sufficiently rapid pace (and OSS systems are considered to evolve rapidly) may be obtaining many of the benefits of problem-space requirements processes through solution-space development processes.

Are Provisionments Advantageous?

We see an increasing trend of rapidly-evolving systems described and reasoned about in terms of whole-system provision- ments, or of component provisionments related through the system's architecture [AAS09, AAS13, ASA10, ScA12a]. This may not only be increasingly typical but also in fact the appropriate approach for reasoning about a stakeholder problem and complex system solution, that is to be implemented by combining complex components. Such an approach manages complexity by reasoning in terms of the capabilities of known (though often themselves complex) components, arranged in architectural configurations in which the capabilities combine to address a problem. It manages ongoing evolution by describing future behavior in terms of differences from past behavior.

Are Provisionments Limited to OSS?

No, they are not; we have seen them in our work as professional CSS developers, most prominently in bug reports and to a lesser extent in feature requests where they serve the same purposes as in OSS.

Some professional CSS developers with whom we have discussed this research report that the requirements they work with might frequently be more accurately described as provisionments. And as we noted earlier in this chapter, OA system development often appears to be guided by reasoning with provisionments, whether the integrators are an OSS project or a proprietary development group, and with good cause.

As we and many other researchers have noted, there is now far more data available from OSS development projects than there is from CSS projects, to which researchers typically have limited or no access. We recall the challenges we have faced in attempting to get access to proprietary development requirements in order to do research. Based on our results so far, we expect provisionments will be found to be in wide use in OSS development, or even in virtually universal use since they align so naturally with reported OSSD processes. It will be more difficult to assess the degree to which provisionments are used in CSS development, but based on what we have learned, we believe their use is widespread there also.

CONCLUSION

In this paper we examined the apparent contradiction between the success of at least some OSS systems and their lack of what may be termed classical requirements artifacts and processes or Classical Requirements. We identified two research questions that are central to this chapter. Here we summarize the answers arising from our study and our examination of related work.

(RQ1) *To what extent do OSS projects in fact use Classical Requirements*? In the data we examined, Classical Requirements was almost completely absent. We found requirements-like artifacts and some requirements-like processes, but virtually nothing exhibiting the three characteristics by which we defined Classical Requirements in an earlier section.

(RQ2) *Where OSS projects do not use Classical Requirements, what artifacts and processes are used instead, if any*? The most prominent requirements-like artifacts we identified were provisionments, statements of features or qualities in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies. These were ubiquitous in the data we examined. The processes were more difficult to characterize; perhaps the most common requirements-like process we saw was the discussion of provisionments in terms of solution-space issues. We hypothesize that architectural reasoning and discussion played a role as well, but did not find strong evidence for it; we may have been looking in the wrong places for that.

In summary, OSS's lack of Classical Requirements results in some of the undesirable outcomes predicted by the broad consensus of software experts and researchers, but not all of them. In some contexts the advantages of OSS appear to outweigh this disadvantage. Further research will be needed to obtain more definitive answers and to provide guidance to making the most effective use of OSS development approaches.

References

[AAS09] Alspaugh, T. A., Asuncion, H. U., and Scacchi, W. (2009). Intellectual property rights requirements for heterogeneously-licensed systems. In *17th IEEE International Requirements Engineering Conference* (RE'09), pages 24–33.

[AAS13] Alspaugh, T. A., Asuncion, H. U., and Scacchi, W. (2013). The challenge of heterogeneously licensed systems in open architecture software ecosystems. In Jansen, S., Cusumano, M., and Brinkkemper, S., editors, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industr*y. Edward Elgar Publishing.

[ASA10] Alspaugh, T. A., Scacchi, W., and 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.

[BCR94] Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The Goal Question Metric approach. In *Encyclopedia of Software Engineering*, pages 528–532. John Wiley and Sons.

[Boe76] Boehm, B. (1976). Software Engineering. *IEEE Transactions on Computers*, 25(12):1126–1241.

[Bro75] Brooks, Jr., F. P. (1975). *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, first edition.

[Bro87] Brooks, Jr., F. P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19. Reprinted from IFIP Congress, Dublin, Ireland, 1986.

[Bro10] Brooks, Jr., F. P. (2010). *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley.

[Cre03] Creswell, J. W. (2003). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, Thousand Oaks, CA, USA, second edition.

[dBv09] de Boer, R. C. and van Vliet, H. (2009). Controversy corner: On the similarity between requirements and architecture. *Journal of Systems and Software*, 82(3):544–550.

[DRW04] Des Rivie`res, J. and Wiegand, J. (2004). Eclipse: a platform for integrating development tools. *IBM Systems Journal*, 43(2):371– 383.

[DSA07] Diallo, M., Sim, S. E., and Alspaugh, T. A. (2007). Case study, interrupted: The paucity of subject systems that span the requirements-architecture gap. In *First Workshop on Empirical Assessment of Software Engineering Languages and Technologies* (WEASELTech'07).

[GaW89] Gause, D. C. and Weinberg, G. M. (1989). *Exploring Requirements: Quality Before Design*. Dorset House, New York.

[Ger03] German, D. M. (2003). GNOME, a case of open source global software development. In *International Workshop on Global Software Developmen*t (GSD'03).

[Jac95] Jackson, M. (1995). *Software Requirements and Specification: a lexicon of practice, principles and prejudices*. Addison-Wesley, Wokingham, England.

[Lam09] Lamsweerde, A. v. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley.

[Mak02] Makuch, E. (2011). *Counter-Strike: Global offensive firing up early 2012*. http://www.gamespot.com/6328645 .

[MFH02] Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346.

[MoF06] Mozilla Foundation (2006). *Firefox2/Requirements*. Mozilla Foundation. http://wiki.mozilla.org/Firefox2/Requirements , accessed 27 Jan 2013.

[Nol07] Noll, J. (2007). Innovation in open source software development: A tale of two features. In Feller, J., Fitzgerald, B., Scacchi, W., and Sillitti, A., editors, *Open Source Development, Adoption and Innovation*: IFIP Working Group 2.13 on Open Source Software, pages 109–120. Springer.

[Nol08] Noll, J. (2008). Requirements acquisition in open source development: Firefox 2.0. In Russo, B., Damiani, E., Hissam, S., Lundell, B., and Succi, G., editors, *Open Source Development,* *Communities and Quality* (IFIP — The International Federation for Information Processing), pages 69–79. Springer-Verlag.

[NoL10] Noll, J. and Liu, W.-M. (2010). Requirements elicitation in open source software development: a case study. In *3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Developmen*t (FLOSS '10), pages 35–40.

[Nus01] Nuseibeh, B. (2001). Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–117.

[Or300] Oreizy, P. (2000). *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution*. PhD thesis, University of California, Irvine. http://www.ics.uci.edu/~peymano/papers/thesis.pdf .

[Pos13] PostgreSQL (2013). *About*. http://www.postgresql.org/about/, accessed 30 March 2013.

[Ray01] Raymond, E. S. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, revised edition.

[Rit84] Ritchie, D. (1984). The evolution of the Unix time-sharing system. *AT&T Bell Laboratories Technical Journal*, 63(6):1577– 1593.

[Sal94] Salus, P. H. (1994). *A Quarter Century of UNIX*. Addison-Wesley.

[Sca02] Scacchi, W. (2002). Understanding the requirements for developing open source software systems. *IEE Proceedings—Software*, 149(1):24–39.

[Sca07] Scacchi, W. (2007). Free/open source software development: Recent research results and emerging opportunities. In *6th Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC/FSE 2007), pages 459–468.

[Sca11] Scacchi, W. (2011). Modding as an Open Source Approac Modding as an Open Source Approach to Extending Computer Game Systems, in *Intern. J. Open Source Software and Processes*, 3(3), 36-47, July-September 2011. Reprinted in S. Koch (Ed.), *Open Source Software Dynamics, Processes, and Applications*, 177-188, Information Science Reference, IGI Global, 2013.

[Sca09] Scacchi, W. (2009). Understanding requirements for open source software. In Lyytinen, K., Loucopoulos, P., Mylopoulos, J., and Robinson, B., editors, *Design Requirements Engineering: A Ten-Year Perspective*, pages 467–494. Springer-Verlag.

[Sca11] Scacchi, W. (2011). Modding as an Open Source Approach to Extending Computer Game Systems, in *Intern. J. Open Source Software and Processes*, 3(3), 36-47, July-September 2011.

Reprinted in S. Koch (Ed.), *Open Source Software Dynamics, Processes, and Applications*, 177-188, Information Science Reference, IGI Global, 2013.

[ScA12a] Scacchi, W. and Alspaugh, T. A. (2012a). Designing secure systems based on open architectures with open source and closed source components. In *International Conference on Open Source Systems* (OSS 2012).

[ScA12b] Scacchi, W. and Alspaugh, T. A. (2012b). Understanding the role of licenses and evolution in open architecture software ecosystems. *Journal of Systems and Software*, 85(7):1479–1494.

[Som04] Sommerville, I. (2004). *Software Engineering*. Addison-Wesley, 7th edition.

[Spi03] Spinuzzi, C. (2003). *Tracing genres through organizations: A sociocultural approach to information design*. MIT Press, Cambridge, MA.

[Sta07] Stallman, R. (2007). *Linux and the GNU system*. http://www.gnu.org/gnu/linux-and-gnu, accessed 30 March 2013.

[TSG94] The Standish Group (1994). *The CHAOS report*.

[VaV00] van Vliet, H. (2000). *Software Engineering: Principles and Practice*. John Wiley & Sons, second edition.

[WiC10] Wiggins, A. and Crowston, K. (2010). Reclassifying success and tragedy in FLOSS projects. In *6th International Conference on Open Source Systems*, pages 294–307.

Chapter 10.

Discussion and Recommendations

Chapter 10.

Discussion and Recommendations

Abstract
This chapter focuses on summarizing and combining the results and recommedations we have developed during our studies in Open Architecture (OA) software systems starting in 2007 through the resent time (early 2015). This chapter thus seeks to bring together what we have learned through our investigations that are presented in the preceding chapters. We recognize that OA software systems include open source software and closed source software elements (components, connectors, configured sub-systems) that are subject to diverse Intellectual Property (IP) obligations and rights, as well as complex cybersecurity requirements. In particular, we draw attention to our current views on how best to align our efforts to address different Better Buying Power initiatives that we believe our efforts can inform and offer guidance. These matters are addressed at the end of this chapter.

What we have learned so far

The relationship between open technology, open architecture, and open source software requirements, and program acquisition is poorly understood. We can call such a view of OSS *product oriented*. Alternatively, we can view OSS as (b) primarily a set of development processes, work practices, project community activities (code sharing, review, modification, 11 redistribution), and multi-project software ecosystem that produce OSS systems and components. This view of OSS as an integrated web of people, processes, and organizations (including project teams operating as virtual organizations [NoS99, CrS02]) is *production oriented* (including production processes, production organizations, production people, and governance over software production [Sca07, SFF06, ScA08, ScJ08]). The requirements for (a) are not the same as for (b), and thus program acquisition targeting (a) may fail to realize the benefits, capabilities, or constraints engendered by (b), and vice versa. As such, there is need to understand how to identify an optimal mix of OSS within OA as both products, and production processes, practices, community activities, and multi-project (or multi-organization) software ecosystem.

The success of DoD's OA and OSS programs in achieving the positive qualities associated with OSS depend on the socio-technical context in which a system is developed and used. The stakeholders and users of an OSS system typically include the developers of that system; they know its goals and requirements implicitly, and can adapt and evolve the system to follow their understanding of the context in which it is used. If DoD is to achieve quick response, rapid adaptation, and context-appropriate use of OSS, it may be necessary to have a representative group of the personnel that are to use and adapt it to the needs they see around them, be OSS developers for that system.

Following from our analysis above, it appears there are a new set of requirements that are emerging that will need to be addressed in any acquisition of a software-intensive system that is stipulated to employ an OA that accommodates OSS components or connectors. Identifying specific requirements for a given program acquisition or system development contract can benefit from consideration of the the following guidelines for how best to realize an OA:

- Determining how much openness is required or desired.
- Identifying guidelines and incentives for software development contractors that encourage them to develop, provide, and distribute/deploy OA systems with OSS components, connectors, and configuration that minimize conflicting OSS license obligations.
- Determining the restrictions, if any, that the OSS licenses used by different software system components, connectors, or configurations within a OA system.
- Identifying alternative OSS component, connector, or configuration candidates that may satisfy a specified overall system architecture.
- Determining scenarios that help reveal whether there are OSS licensing conflicts for a given set of OSS components, connectors, or configuration.
- Identifying and analyzing any OSS licensing obligations that must be satisfied for the resulting system to be available for redistribution.
- Identifying and validating OSS license conformance criteria for configured systems intended for redistribution.

Further elaboration on these guidelines is subject to additional research, application, and refinement. However, they do provide a useful starting point for discussion, debate, and action in program acquisition.

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 [Sca07], components will be patched and upgraded (perhaps with new license constraints), and inter-component connections will be rewired or remediated with new connector types. As such, sustaining the openness of a configured software system will become part of ongoing system support, analysis, and validation. This in turn may require ADLs to include OSS licensing properties on components, connectors, and overall system configuration, as well as in appropriate analysis tools [cf. BCK03, MRT99].

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

Advances in the identification and extraction of configured software elements at build time, and their restructuring into architectural descriptions is becoming an ever more automatable endeavor [cf. ChS90, KaC99, JBA08]. Further advances in such efforts have the potential to automatically produce architectural descriptions that can either be manually or semi-automatically annotated with their license constraints, and thus enable automated construction and assessment of build-time software system architectures.

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

Subsequently, 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.

Moving forward, at least two topics merit discussion following from our approach to semantically modeling and analyzing OA systems that are subject to heterogeneous software licenses. One is how our results might shed light on software systems whose architectures articulate a software product line, while the other is how our approach might be extended to also address the semantic modeling and analysis of *software system security requirements*.

Organizing and developing software product lines (SPLs) relies on the development and use of explicit software architectures [Bos00, ClN01]. However, the architecture of a SPL is not necessarily an OA — there is no requirement for it to be so. Thus, we are interested in discussing what happens when SPLs may conform to an OA, and to an OA that may be subject to heterogeneously licensed SPL components. Three considerations come to mind. First, If the SPL is subject to a single homogeneous software license, which may often be the case when a single vendor or government contractor has developed the SPL, then the license may act to reinforce a vendor lock-in situation with its customers. One of the motivating factors for OA is the desire to avoid such lock-in, whether or not the SPL components have open or standardscompliant APIs. Second, if an OA system employs a reference architecture much like we have in the design-time architecture depicted in Figure 1, which is then instantiated into a specific software product configuration, as suggested in the build-time architecture shown in Figure 2, then such a reference or design-time architecture as we have presented it here

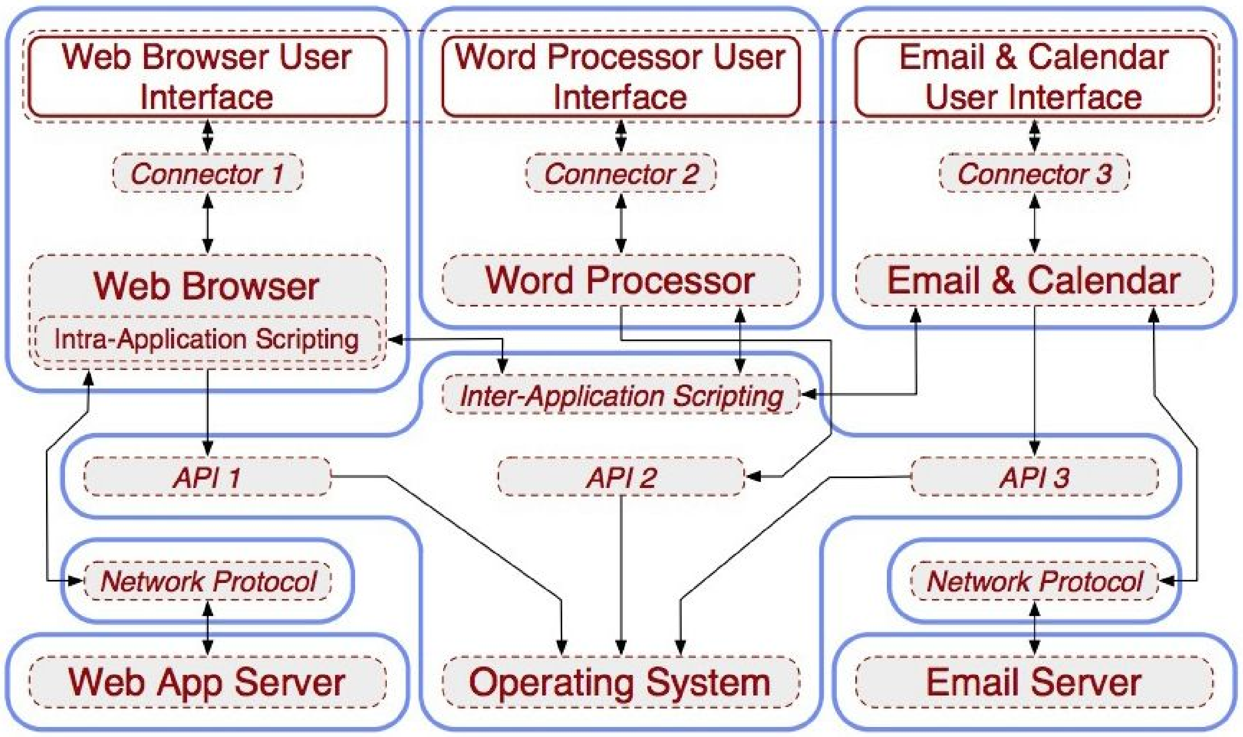


Figure 1. An example design-time OA system view, including software component and connector types, within specified cybersecurity containers.

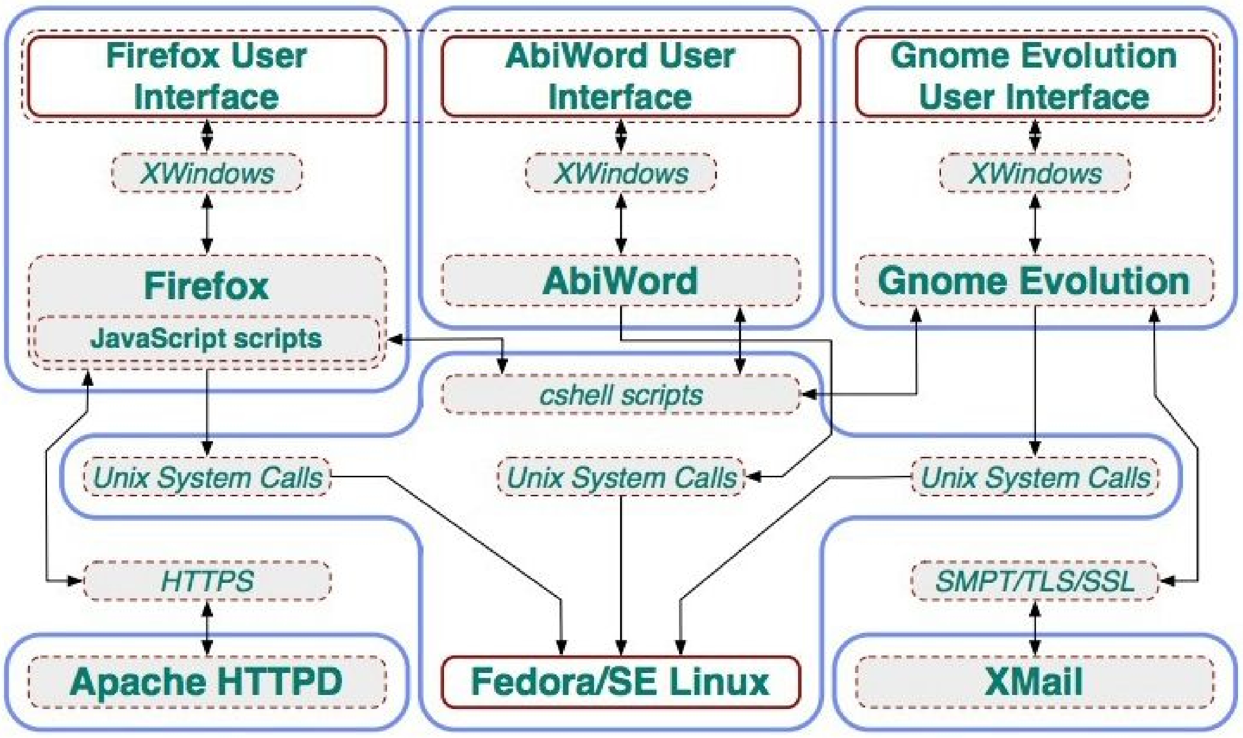


Figure 2. An example integration and test build-time OA system view, including software component and connector instances, within specified cybersecurity containers.

effectively defines a SPL consisting of possible different system instantiations composed from similar components instances (e.g., different but equivalent Web browsers, word processors, email, calendaring applications, relational database management systems). Third, if the SPL is based on an OA that integrates software components from multiple vendors or OSS components that are subject to heterogeneous licenses, then we have the situation analogous to what we have presented in the preceding chapters, as well as in this chapter. So SPL concepts are compatible with OA systems that are composed from heterogeneously licensed components.

Next, as already noted, software licenses represent a collection of rights and obligations for what can or cannot be done with a licensed software component. Licenses thus denote non-functional requirements that apply to a software systems or system components as intellectual property (IP) 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 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 [BrA05, BrA08].

Traditionally, developing robust specifications for non-functional software system security properties in natural language often produces specifications that are ambiguous, misleading, inconsistent across system components, and lacking sufficient details [YaC06]. Using a semantic model to formally specify the rights and obligations required for a software system or component to be secure [BrA05, BrA08, YaC06] 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.

The approach we have been developing for the past few years for modeling and analyzing software system license architectures for OA systems [AAS09, AAS13, ScA08], may therefore be extendable to also being able to address OA systems with heterogeneous “software security license” rights and obligations. Furthermore, the idea of common or reusable software security licenses may be analogous to the reusable security requirements templates proposed by Firesmith [Fir04] at the Software Engineering Institute. Consequently, such an exploration and extension of the semantic software license modeling, meta-modeling, and computational analysis tools to also support software system security can be recognized as a promising next stage of our research studies.

Our approach to specifying and analyzing the security requirements for a complex OA system is based on the use of a security license. As noted, a *security license* [AIS12] is a new kind of information structure whose purpose is to declare operational capabilities that express the obligations and rights of users or program to access, manipulate, control, update, or evolve data, control signals, and accessible software system elements. Our proposed security license is influenced by IP licenses that are employed to specify property control and declared copyright freedoms/restrictions, such as those for OSS components subject to licenses like the GPLv2, MPL, LGPL, or others. These IP licenses as information structures often pre-exist to facilitate their widespread use, dissemination, and common interpretation. Further, the choice of which IP license to choose or assign to a software component results from a trade-off analysis typically performed by the components producers, rather than the system integrators or consumers, as a way to protect or propagate the obligations and rights to use, evolve, and redistribute the updated component's open source code.

The security licenses we propose may or not necessarily exist prior to their specification and assignment to a given OA system. Similarly, we may anticipate or expect that generic security licenses will emerge and be assigned by software component producers, as they have for OSS components, though no such security licenses from producers yet exist. However, one follow-on goal we seek to address is whether and how best to specify security licenses for different types of software elements or components so that it becomes possible to semi-automatically specify the security license for a given component or composed OA system through the reuse and instantiation of security requirement templates. This idea is somewhat similar to the license templates and taxonomy that is employed by the Creative Commons for non-software intellectual property like online art or new media content (cf. http://creativecommons.org/licenses/). In this regard, it may be possible to develop a technique and supporting computational environment whereby system integrators or consumers can conveniently specify the security requirements they seek (e.g. fill out online security requirements forms), while the environment interprets these specifications to generate operational security capabilities that can be guard the entry and exit of data or control information from the appropriate containment vessel that encapsulates the corresponding system element. Consequently, this is a topic for further study and investigation.

Next, one might wonder why it is not simply desirable to have maximum system security under all circumstances. When considering the alternative run-time system composition variants shown in Figure 2, it appears there may be trade-offs in one layout of security capabilities over another. For example, if the layout in Figure 2 were revised so that each OA software element (i.e., each component and connector instance) this potentially maximizes security by encapsulating each system element within its own containment vessel. This in turn requires a VM technology of a kind different from that commonly available (e.g., like VMware), and instead requires a new lightweight VM technology that can provide security capabilities (e.g., create, read, update authorizations) for potentially smallscale software elements (e.g., Cshell inter-application integration or run-time scripts). Similarly, the different security containment layouts may affect system performance, ease of evolutionary update, and associated level of security administration. For example, if all top-row softwre components were running on a single personal computer, it might be the choice to simplinstall a virtual machine on the PC, so that all local components and connectors are containied within a single virtual machine. This would be a simpler to install and administer security container scheme, though providing a leser level of security, which might be acceptable for PC users at home. But these again all represent trade-offs in the desire to achieve affordable, practical, and evermore robust and testable secure software component/system capabilities build-time and run-time. Thus, we take the position that it is better to provide the ability to specify and analyze the security requirements of different software elements at designtime, as well as specify and analyze the security capabilities at buildtime and run-time, rather than the current practice that does not account for system architecture nor license architecture, and is thus inherently vulnerable to attacks that can otherwise be prevented or detected.

One other topic that follows from our approach to semantically modeling and analyzing OA systems that are subject to software security licenses. More specifically, how our approach and emerging results might shed light on software systems whose architectures articulate a software product line.

Accordingly, organizing and developing software product lines (SPLs) relies on the development and use of explicit software architectures [Bos00, CIN01]. However, the architecture of a secure SPL is not necessarily a secure OA — there is no requirement for it to be so. Thus, we are interested in discussing what happens when SPLs may conform to a secure OA, and to an OA that may be composed from secure SPL components. Three considerations come to mind.

First, if the SPL is subject to a single homogeneous security software license, which may often be the case when a single vendor or government contractor has developed the SPL, then the security license may act to reinforce a vendor lock-in situation with its customers. One of the motivating factors for OA is the desire to avoid such lock-in, whether or not the SPL components have open or standards-compliant APIs.

Second, if an OA system employs a reference architecture much like we have in the design-time architecture depicted in Figure 1, which is then instantiated into a specific software product configuration, as suggested in the build-time architecture shown in Figure 2, then such a reference or design-time architecture as we have presented it here effectively defines a SPL consisting of possible different system instantiations composed from similar components instances (e.g., different but equivalent Web browsers, word processors, email, calendaring applications, relational database management systems).

Third, if the SPL is based on an OA that integrates software components from multiple vendors or OSS components that are subject to heterogeneous security licenses (i.e., those that may possible conflict with one another), then we have the situation analogous to what we have presented in this paper. So secure SPL concepts are compatible with secure OA systems that are composed from heterogeneously security licensed components.

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 [ScA08, ScA11].

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 using the kind of system security licenses we anticipate [ScA11], 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.

How best to improve and streamline acquisition processes for secure OA systems

The transition to the development, deployment, and sustainment of software-intensive systems based on an OA means that new or revised acquisition processes may be needed. In particular, we believe such advances call for (a) the adoption of open business models within DoD and its industry partners, (b) open source approaches to creating Web-based acquisition processes [Sca01] that specifically address BBP initiatives, and (c) employing techniques for streamlining these processes [ChS01, Nis98, Sca01, ScN97] for secure OA systems. Each is described in turn in this section.

Encourage the adoption of acquisition business models in open source formats

One goal of BBP initiatives is to reduce costs by improving competition. Such a situation may be disconcerting to legacy software producers who are long experienced with the long-term development of proprietary, large-scale software systems with closed architectures that are subject to traditional, cumbersome, and costly software product licenses and license management regimes [And12, Ko09]. A move towards agile and adaptive development of secure OA systems based on software components, that can be developed/integrated more rapidly and at lower cost with more favorable IP licenses, represents a new acquisition strategy [RBC12, ScA13b]. This suggests the need to incentivize software producers and system integrators, so as to insure their ability to effectively produce both proprietary and OSS components that are economically viable yet cost effective to the Government over the life of such systems. The overall BBP mandate recognizes this situation, but does not specify the means for how best to accomplish it. We believe one promising candidate is for Defense Enterprises and Program Offices to adopt new open business models.

The business models we have in mind should be rendered in an open source format. Such models should be computer-processable (i.e., amenable to automated enactment support) and transparent to participants in the acquisition workforce (e.g., available through Web-based application systems [Sca01, ScN97]). They should be similarly open to participants in software producer, system integrator, and system user enterprises. These models should incorporate a product line of common/reusable open system architectures that can integrate functionally similar software components in order to realize domain-specific system solutions (e.g., for domains like command and control, weapon systems, or enterprise computing) [BeJ10, GuC10, JoB11, RBC12, ScA12b, SEI07, WoS11]. These business models should incorporate Web-based computational models of acquisition processes [Nis98, Sca01, ScN97] that manage the system development and support processes that surround the OA product line system models. Finally, these business models should highlight which acquisition or system development processes, or OA system features, require attention to IP licenses.

Prior research has demonstrated that significant cost reductions and process streamlining are possible when open source business process models are utilized [ChS01, Nis98, ScN97, Sca01]. These kinds of models can be subjected to performance measurement across multiple acquisition process enactments, continuous improvement, and process redesign by the acquisition workforce [Sca01]. Now we propose to enhance and extend their value through the incorporation of OA system models. While demonstrating such a capability is beyond the scope of this study, prior research results suggest the plausibility of such an approach. So future acquisition research targeting BBP may be directed to creation of open business models that can be openly accessed, reused, modified, and redistributed where appropriate.

Encourage the development, (re)use, and refinement of open source models of acquisition processes

As noted, prior research has demonstrated the value and real payoffs of Web-based computational models for Defense acquisition processes [ChS01, Nis98, ScN97, Sca01]. However, many technological advances, organizational transformations, and shifting Defense priorities have occurred since these results were first demonstrated and deployed years ago. Our own studies on design of secure OA system product lines are an example of technological advances not addressed in our earlier process models. But without explicit, open source process models that can be enacted through Web-based user interfaces (i.e., Web browsers accessing remote application services while tracking process enactment progress and performance parameters), then the ability to realize their benefits like process streamlining and cost reduction are elusive and difficult to manifest. Among the reasons for why this is so includes overcoming gaps for how best to: (a) monitor and measure acquisition process performance without automated enactment support; (b) redesign legacy processes to better accommodate technical advances and to remove ineffective bureaucratic procedures, or that transform acquisition processes in ways that do more with less while also empowering the acquisition workforce; (c) design new acquisition processes like those for acquiring secure, component-based OA software systems subject to multiple IP licenses; and (d) accommodate software IP licenses and license management regimes as acquisition process cost elements. To better understand what gaps exist in these four areas, we now describe techniques for streamlining the acquisition processes for secure OA system.

Develop and employ techniques for streamlining acquisition processes for secure OA systems
A goal of this paper is to identify ways and means for streamlining acquisition processes for secure OA systems. In particular, we focus on four kinds of techniques that can be used to streamline such processes in ways that are responsive to the BBP initiative for open system architectures subject to complex IP licenses. These techniques are illustrative rather than exhaustive, as other kinds of techniques in other areas are also expected to exist and be available for practice by the acquisition workforce.

- *Acquisition Process Measurement and Assessment* – The most direct way to determine the efficiency and effectiveness of acquisition processes is by measuring their structural attributes. Such attributes indicate things such as (a) length of longest path of process steps/actions (process length); (b) number of distinct process paths (process width); (c) number of sub-process levels (process depth); (d) total number of process steps (process size); and (e) process size divided by process length (process parallelism), and others metrics [Ni98]. But without an explicit graph-based model of acquisition processes, such measurements are impractical or implausible. Nonetheless, such metrics are a key for where to look for process improvement or process redesign opportunities. One might also recognize that some acquisition processes are underspecified, for example, by not explicitly accounting for where software licenses are negotiated or license trade-off analysis done. Similarly, as OA systems may include software components subject to different licenses [ASA10], then how are componentcomponent license interactions assessed or analyzed, if at all? If acquisition processes do not explicitly account for new acquisition or license management activities that emerge due to advances in OA system development, then such processes are underspecified, which means their costs are hidden and difficult to control/minimize. Thus, if the goal of BBP is to help improve the affordability of OA systems within the DoD, then we need to be able to systematically model, measure, and assess our acquisition processes [Sca01]. Similarly, we need to better understand how to measure and assess open business models for use within DoD and its industry partners to incentivize and continuously improve competition and Defense affordability

- *Acquisition Process Redesign and Evolution* – Once we have the ability to measure and assess current/emerging acquisition processes for secure component-based OA systems, we can then begin to analyze (or simulate) them in ways that reveal process redesign opportunities and transformation heuristics [ChS01, Nis98, ScN97, Sca01]. Among the acquisition process pathologies we seek to identify are those where measured processes reveal sub-processes with low effectiveness (indicating high levels of iterative rework), low efficiency (indicating slow or bureaucratically cumbersome process steps that add marginal value to process completion), and problematic sub-processes (indicating underspecified process steps, steps that generate processing delays due to missing/or incorrect acquisition data, or inappropriate automated process enactment support). For example, current processes that assume long-term acquisition of monolithic software systems with proprietary components integrated within a closed architecture, are likely not well-suited to address the challenges for acquiring secure OA systems that integrate software components from different online repositories. We also place our acquisition workforce at a disadvantage if we do not empower them with the ability to measure, assess, and adaptively redesign their processes as technological advances like component-based OA systems are to be acquired. New software component technologies and software ecosystem niches [ScA12a] are also emerging which necessitate new continuous development processes and new license management practices, and thus redesign/evolution of acquisition processes [ScA13a, SBN12]. These examples all point to new opportunities to redesign, evolve or other transform existing acquisition processes to better fit the challenges posed by the development, deployment, and support of secure, component-based OA systems. Finally, we can empower the acquisition workforce to realize continuously improved acquisition processes if we can provide them with the training and resources for modeling, analyzing, and redesigning their acquisition processes in ways that empower them to utilize Web-based automated process enactment systems, which also allow them to try out and walkthrough alternative process redesigns before committing to their use in daily operations.

- *Design New Acquisition Processes* – Across the DoD community, there are many variations in practice for how to specify and model the architecture of a softwareintensive system. Some practices focus attention primarily on identification of major components or abstract layers, while minimizing (or ignoring) attention to interfaces and interconnections, which are more challenging to identify and manage. However, the BBP initiative for OA systems points to the need for managing explicit interface specifications that identify and reinforce the use of standard interfaces [DAU12]. Without such interface and interconnection specifications, it is not possible to determine the scope or potential conflicts/matches between the IP licenses (and thus TD rights) for the overall system architecture. In contrast, we have demonstrated in our prior research that component-based OA systems become tractable and evolvable from IP license management and security perspectives when the system architecture of components, connectors, and interfaces are explicitly modeled [ASA10, ScA11, ScA12a, ScA12b, ScA13b].

The use of standard interfaces allows for simpler renderings of OA system structure, and thus simplifies license analysis. Further, once interfaces and interconnections become explicit, software component producers, system integrators, and/or system consumers can determine/negotiate which interfaces should be standardized in order to improve competition and affordability. These standards may then define acceptable data types, relationships between data types, data attribute value ranges, and exceptional data values in ways that are open, sharable, and reusable, as well as extensible when appropriate. Such improvements become possible by enabling an agile, adaptive ecosystem for software components of different size and capability relative to OA system product lines for different application domains [RBC12, ScA12a, ScA13b]. Therefore, another important technique for streamlining the acquisition of secure, component-based OA systems, in line with BBP initiatives, is to provide the acquisition workforce with the resources and automated support to design and computationally enact new acquisition processes (i.e., explicitly modeled processes [ChS01, Nis98, ScN97, Sca01]), where the processes are open, agile, and adaptive. Such modeled processes may also then be shared, reused, continuously improved, and redistributed across the ecosystem of Defense Enterprises and Program Offices.

- *Cost Management as an Acquisition Process Design Element* – Part of the promise of the move to OA systems stems from their perceived potential to reduce acquisition life cycle costs, improve competition, and improve Defense affordability [DAU12]. But where and how are the associated cost factors or cost drivers for OA systems identified, tracked, and managed? After all, if we do not know where the cost factors are, or what activities, conditions, or events drive OA system acquisition costs, then we cannot effectively control such costs, nor make well-informed system capability/cost tradeoffs. For example, people who manage the acquisition of large-scale software systems within various Defense Enterprises are familiar with the many types of end-user license agreements for proprietary, closed source software systems [And12]. In contrast, these people may not know how best to manage the acquisition of OA systems whose software components are jointly subject to different OSS or proprietary licenses.

The acquisition workforce has also learned in practice that software IP licenses are subject to change over time. However, one consequence is that long-lived or widely used software systems become more costly and much less amenable to technology substitution or vendor replacement, thereby reducing competition due to vendor lock-in. This works against Defense affordability. In contrast, emerging online repositories offer different kinds of software component with different functional capabilities (described earlier), along with different IP licenses and end-user licenses (e.g., low cost, per user licenses). These repositories of software components represent a means for increased competition and affordability, but subject to different acquisition, development or integration processes that are just coming to light. Accordingly, we believe that streamlining the acquisition process for secure, component-based OA systems requires that IP license cost obligations (e.g., license fees for end-user agreements) and license management regimes need to be incorporated into: process measurement and assessment, process redesign and evolution, and design of new acquisition processes. This is also a subject for further acquisition research, but one offering practical nearterm consequence.

Achieving Better Buying Power Goals

Better Buying Power (http://bbp.dau.mil/) is part of DoD's mandate to *do more without more* by implementing best practices in acquisition. BBP identifies seven areas of focus that group a larger set of a few dozen initiatives that offer the potential to restore affordability in defense procurement and improve defense industry productivity. One of the seven areas focuses on promoting or increasing competition, and this area includes an initiative to “enforce open system architectures and effectively manage technical data rights” [DAU12]. Technical data rights pertain to two categories of Intellectual Property (IP): they refer to the Government's rights to (a) *technical data* (TD – e.g., product design data, computer databases, computer software documentation); and (b) *computer software* (CS – e.g., source code, executable code, design details, processes, and related materials). These rights are realized through IP licenses provided by system product or service providers (e.g., software producers) to the Government customer, so long as the customer fulfills the obligations stipulated in the license agreement (e.g., to indicate how many software users are authorized to use the licensed product or service according to a fee paid).

As already noted, our acquisition research has focused on issues addressing OA systems and IP licenses since 2008 [ScA08], as well as forward to the acquisition of secure OA systems for command and control (C2) and enterprise information systems [ScA11, ScA12b, ScA13b], where security requirements can be expressed in a manner similar to IP obligations and rights. Therefore, here we turn to identify how a sample of different goals of the BBP initiatives interact or relate to the trends and challenges examined so far in this paper. Representative BBP goals are highlighted, then followed by a brief examination.

- *Increase competition* – One central purpose for acquiring OA systems is to increase the likelihood of competition among system producers who can provide software components that can be replaced by similar offerings by other component producers. We demonstrate how this can work when system architectures are explicitly modeled, and their software components and interconnections are similarly specified in an open manner [AAS13, ScA12a].

- *Adopt OA systems that utilize standardized interfaces* – Open system architectures that can accommodate common components from alternative producers require that components utilize standardized interfaces, whether in the form of: open Application

Program Interfaces (APIs); standard data exchange protocols; or standard data representations, formats, and meta-data [ScA08]. But also noted earlier, app and widget components at present have a plethora of standardized interfaces, and it is unclear which will survive, be sustained, be widely adopted (inside/outside of DoD), and be evolved [End13a].

- *Utilize open source software components where appropriate to reduce costs* – another aspect of openness that OA systems embrace and DoD policy accepts is to utilize system components developed as open source software (OSS) [DIS12]. Utilization of OSS components, along with composing OA systems that incorporate OSS and closed, proprietary components, does require careful attention to the management and analysis of multiple IP licenses that apply to different OA system components, as well as determining what overall IP and/or cybersecurity rights and obligations apply to the overall system [AAS13, ScA12a], especially for C2 systems [AAS13, ScA13b, ScA13c].

- *Increase small business roles and opportunities* – one way to increase competition in the realm of OA systems is to identify where smaller scale software applications (apps) or widgets can be utilized, which might be produced by small businesses or startup ventures which dominate much of the online markets for Web-based or mobile device apps/widgets. Small businesses may further be advantaged by their utilization of OSS infrastructure components, platforms, or remote services, since large commercial contractors may not see sufficient profit margins to develop proprietary alternatives. So OAsystems that accommodate OSS components that can integrate custom apps/widgets into innovative system capabilities (C2SC), may then realize new opportunities for DoD customers. Other small business opportunities may similarly arise for such ventures that focus on emerging cybersecurity assessment or tool development services.

- *Use technical development phase for true risk reduction and rapid prototyping* – In looking forward, there is potential interest in seeing the BPP initiative evolve to also address risk as an implicit cost driver. This might allow or innovative ways and means to reduce emerging risks through accelerated or “look ahead” system acquisition and development approaches that emphasize increased reliance on rapid prototyping. This kind of rapid prototyping might even be performed by appropriately trained end-users or warfighters. A move towards OA systems for Web-based and mobile devices that rely on apps/widgets retrieved from online marketplaces, that can be composed through interpretive software program “scripting” and mashup techniques, is a clear example of this [End13, GMH13 GuW12, ScA13a]. Thus, it is not surprising to find such emerging techniques being investigated and assessed for possible production of new C2 capabilities [GBC14, GMH13, ScA13b].

Doing more without more – an overall summary of the current BBP initiative is focusing attention of how to make acquisition more agile, to do more without more, and to develop a new generation acquisition workforce that can enact acquisition processes that are thin and flexible when needed, yet robust and cost-effective, while also being amenable to continuous improvement. This is indeed a real challenge to fulfill, and beyond the scope of what current acquisition practices are likely to achieve without targeted investment in acquisition improvement research. To be clear, one just needs to consider emerging opportunities (and potential asymmetric cybersecurity threats) that arise through the desire to develop next-generation C2SC that are to be composed from apps/widgets that can operate on Web-based/mobile devices. What are the best processes or practices for acquiring, developing, and sustaining deployed systems that are to be built using these new software technologies (e.g., apps/widgets for mobile devices)? How should these processes and practices be adapted to accommodate personal devices (e.g, Apple iPhones, Android tablet, Microsoft Mobile Phone, Blackberry 10 phone) that individual warfighters, joint force troops, or contracted service providers bring with them into the battlespace? How must acquisition processes be best adapted to accommodate and rely on software supply chains that arise around consumer-oriented app marketplaces as possible ways/means for *doing more* (e.g., rapidly prototyping warfighter composable C2 app/widget mashups [GMH13]) *without more* (e.g., warfighters who bring their own mobile computing devices for use in C2 contexts) [GBC14]? Once again, these are critical questions to address and resolve through new acquisition research and supporting technology development.

Emerging Challenges in Achieving BBP through OA Software Systems

The business models and IP licenses for software components are tightly coupled: software component licenses codify component producer business models. Said more simply, licenses codify business models. So different software business models imply different software license obligations and rights, and different license types reflect different possible business models. Licenses are generally recognized as contracts regarding IP expressed through terms and conditions that specify obligations and rights stipulated by the component's producer to enable/constrain what can be done with the component by its integrator or end-users. Understanding and assuring software IP obligations and rights is iroutinely a task for acquisition management, and thus a task to be competently performed by the acquisition workforce

Obligations (like purchase costs/fees paid, or to insure access to open source software code modifications) denote conditions, events, or actions imposed by a software producer (the licensor) that must be fulfilled by the software integrator/customer enterprise (the licensee) in order to realize the *rights* granted or withheld by the licenses (right to use; right to distribute copies; no right to distribute modified copies, etc.). Note that software system integrators play a role is shaping the obligations and rights imposed on customer enterprises based on choices they make in how software component-based systems are designed, built, and deployed. So where/who does system integration occurs matters, as does whether customer enterprises that acquire systems have policies that determine which software licenses (or business models) they will accept.

Similarly, we note that “cybersecurity requirements” can also be expressed and analyzed in terms of obligations and rights [ScA11, ScA12b]. This suggests the the problems and solutions to software IP license management will be similar in kind or form to those for cybersecurity assurance. Below, we just focus attention to software IP obligations and rights, though the same consequences may apply to the cybersecurity of OA systems and components.

There are many unstated consequences that can arise when software licenses are not well understood. Here are some examples we have seen within the DoD context.

- *Different military services specify which software licenses they do and do not accept*. This can give rise to service **X** refusing to use any software component subject to license ***A*** (e.g., GPL—Gnu Public License), while service **Y** deploys mission-critical command and control systems that incorporate components subject to license ***A*** [ScA08]. This may imply that service **X** will not allow connection of its C2 systems to service **Y** C2 systems, which can readily look like a bad outcome.

- *Acquisition program managers/staff (including in-house legal counsel) may not understand how software licenses affect OA system design, and vice-versa.* Component-based system design can determine which software licenses will fit, or which can fit if the system design is altered to encapsulate desirable software components with somewhat problematic license obligations or rights [ScA13a].

- *Software license obligations and rights propagate through system development life cycle activities in ways not well understood by system developers, integrators, end-users, or acquisition managers*. We have investigated and described many examples of this in a recent paper [ScA13a] that shows how license constraints are mediated by software system design, build-integration, deployment, post-deployment support tools and activities.

- *Different acquisition programs within DoD and other government agencies may independently reinterpret software component licenses.* This realizes enterprise-wide inefficiencies, as well as increases avoidable costs. It appears to be technically possible to codify software component licenses by type or producer, especially with regards to performative obligations and operational rights that Program Offices or customer organizations seek. The license modeling techniques we have investigated demonstrates the potential, practicality, and scalability of such possibility [AAS13, ScA12a, ScA12b, ScA13b]. However, it may be most efficient and most effective for DoD to have common legal interpretations for different licenses (or different business models). Such interpretations could be common, if produced by a central legal authority (e.g., Office of General Counsel). Alternatively, it may also be possible for DoD and other government agencies to provide an open framework or (acquisition) policy guidance whose purpose is to encourage software producers to not only provide software licenses in current narrative forms, but also to provide them in computer processable forms (using domain-specific languages) amenable to automated license analysis. Once again, this is a form of guidance and training we can provide, but it is not one that we can impose on anyone. We believe it is in the best interest of DoD and other government agencies to employ software licenses that are both human readable and formally processable though automated means, at least in terms of software license obligation and right determinations.

- *Failure to understand software license obligation and rights propagation can reduce DoD buying power, increase software life cycle costs, and reduce competition.* Guidance from the OUSD for Acquisition, Technology, and Logistics recommends programmatic adoption of different Better Buying Power initiatives grouped into seven focus areas of relevance (http://bbp.dau.mil/sevenareas.html) as programmatic methods for doing more without spending more. Acquiring licensed software components is a cost-generating activity, whose costs/fees can be reduced while acquiring evermore agile and adaptive software components and open architecture component-based systems. However, software license non-compliance or worse, infringement, on the part of DoD will generate costs, program delays, as well as reduce agility and adaptation, all of which can be avoided. Such situations can and must be avoided through acquisition and development practices with little/no additional cost to affect. Such practices can be codified within open source business processes or open source computational business process models that can be shared, customized to specific program needs, redistributed and archived [ScA13b].

- *Software producers often provide idiosyncratic licenses that generally conform to common business models and common license types.* This seems mainly to arise from efforts by software producers to protect or update their business models in ways that improve their financial yield or protect/lock-in their customer base. This in turn generates demand for time, attention, and effort from legal counsel that support acquisition programs, while also reducing the effectiveness and timeliness of program acquisition efforts. DoD and other government agencies may be able to explicitly specify in advance what kinds of generic software license obligations they will accept and what kinds of generic software rights they seek, through their own explicit business models. Such specifications can be codified and provided to software producers in open source manner through software license acquisition policies. Software producers might then separate license terms and conditions that do and do not address current license acquisition policies, in order to streamline licensing design and analysis practices for the mutual benefit of software producers, integrators, and customers.

- *Software producers generally provide software licenses that are assumed to legally dominate in systems composed of components from different software producers or integrators.* We refer to software systems (or systems of systems) composed from components (e.g., apps, widgets) subject to different licenses as "heterogeneously-licensed systems" (HLS) [ASA10, AAS13]. Popular Web browsers that are compatible with widgets, apps, or plug-in components (e.g., Google Chrome, Mozilla Firefox) are subject to dozens of component licenses. Popular COTS software components also sometimes encompass components subject to multiple licenses. In both situations, the component producer asserts overall component license obligations and rights in ways that are compatible with the licenses included therein (or so we hope). But when we deploy components that are composed into complex system architectures, or employ components that support on-demand download and implicit integration of smaller components (widgets, plug-ins, scripts, etc.) from online stores, then analysis of license obligation and rights propagation or encapsulation matters. Such technical details can readily overwhelm program acquisition managers and legal staff, thereby reducing the agility and adaptation of component-based system development or deployment. Provision of automated license analysis capabilities within software license management systems should be able to overcome this situation.

- *Given the challenges of HLS, it is unclear what kinds of trade-offs can/should be addressed by software system integrators or program acquisition staff to maximize overall system development agility and evolutionary adaptation.* This situation is not unique to DoD, but is in fact widespread. However, as DoD and other government agencies move to embrace agile and adaptive component-based software systems to realize new, more timely system capabilities at lower cost compared to legacy approaches, then there is need to provide guidance for how to identify and manage such trade-offs. Failure to recognize the challenges of analyzing and managing HLS systems translates into opportunities lost while avoidable costs increase. We can and should do better than this. But this will require that resources be allocated to identify, articulate, train, and iteratively refine best practices about how, where, when, and why these trade-offs arise. Such knowledge should therefore be captured, codified, shared, accessed, updated, and redistributed in an open source manner.

- *Software IP license and cybersecurity obligations and rights must be tracked, accounted, and managed.* A move to component-based open architecture systems increases organizational overhead for managing software licenses. This overhead can be reduced, or better transformed into productive, value-adding business practices, through use of automated software obligations and rights management systems (SORMS). While Software License Management Systems exist and are routinely used by software component producers (to keep track of who has a licensed copy of their software products), SORMS do not exist at this time for software system integrators or customer enterprises.

- *DoD and other Government agencies would financially and administratively benefit from engaging the development and deployment of an open source automated SORMS.* This may represent the lowest cost means for simplifying license analysis while maximizing the benefits of agile and adaptive component-based software systems acquisition within the DoD and other government agencies. SORMS can help to better

DoD software buying power. Similarly, an open source SORMS would also be of value to smaller or startup software producers who may best be able to   create innovative and agile software components (widgets) in cost-competitive ways. Last, an open source SORMS intended for software integrator/customer enterprises would be of value to large, established DoD software producers, as a medium through which   larger-scale software component acquisitions (e.g., components acquired for standardized deployment throughout an enterprise can be negotiated and simplified.

Finally, as suggested along the way, *all* of these consequences can be both anticipated and mitigated through action and careful investment in enabling solutions.

New Practices to Realize Cost-Effective Acquisition of OA Software Systems for Web-Based and Mobile Devices

The trends and concerns identified above point to substantial challenges in identifying what can be done to both realize cost-effective BBP for Web-based and mobile device software apps, and to do so in ways that enable and empower the acquisition workforce in the years ahead. Technology, better buying practices, new business models, and new cybersecurity requirements all point to the need for future research and development of new acquisition support technologies, work processes, and guidance practices. The goal is to make sure that acquisition time and effort does not become the main cost and the main risk factor going forward on the path to agile OA Web-based or mobile compatible C2 system development, deployment, and sustaining system evolution.

At this point, we see at least three key areas of opportunity for future acquisition research and development. First, we need to research and develop ***worked examples*** of well-formed OA system architectures that are appropriate for C2 system capabilities, and that accommodate Web-based apps, widgets, and mobile devices. Such OA system architectures should specify representative and standardized component interfaces. The examples should also include carefully specified shared agreements that account for different IP licenses and diverse business models of software producers, system integrators, and multiple end-user organizations who must collectively act in ways that enable agile development and adaptive evolution of demonstrable C2 system capabilities.

Second, we need robust ***open source models*** of application security processes and reusable cybersecurity requirements that account for exigencies in heterogeneous app/widget software ecosystems, account for software evolution dynamics, formation and continuous improvement of automation-compatible shared agreements, and more. These models should account for description of current process practices, prescription of required verification and validation activities and outcome (deliverable documents or online artifacts), and proscription of what tools/techniques to use, by whom, when, where, and how.

Third, we need reasonably precise, human readable and computer processable ***domain specific languages*** (DSLs) for specifying, and automated analysis tools for continuously assessing and continuously improving, cybersecurity and IP license requirements for dynamically evolving Web/mobile C2 systembased capabilities. The DSLs needed must be able to specify and operationalize the shared agreements between different DoD organizations, government agencies, and commercial enterprises involved in producing, integrating, or evolving component-based OA C2 system capabilities.

Overall, what we call for is similar in kind to what we have already produced and applied in other software development domains, using then current technologies [JeS05, ScA08]. What we now call for is a reinvention and repurposing of these concepts, but in contemporary forms scaled and secured in ways that best meet the needs of the DoD program offices, acquisition program managers, and others in the acquisition workforce to best support BBP 3.0 initiatives for Web-based and mobile device software components (widgets, apps, plug-ins).

Conclusions

The DoD, other government agencies, and most large-scale business enterprises continually seek new ways to improve the functional capabilities of their software-intensive systems. The acquisition of OA systems that can adapt and evolve through replacement of functionally similar software component applications (apps) and widgets is an innovation that can lead to lower cost systems through more agile system development and adaptive system evolution. Our research identifies and analyzes how new software component apps and widgets, their IP license and cybersecurity requirements, and new software business models can interact to drive down (or drive up) total system costs across the system acquisition life cycle. The availability of such new scientific knowledge and technological practices can give rise to more effective expenditures of public funds and improve the effectiveness of future software-intensive systems used in government and industry.

Our study reported here also identifies a new set of technical risks that can dilute the cost-effectiveness of Better Buying Power efforts. It similarly suggests that current acquisition practices aligned with BBP can also give rise to acquisition management activities that can dominate and overwhelm the costs of OA system development. This adverse condition can arise through app/widget vetting, new software business models, opaque and/or underspecified acquisition management processes, and the evolving interactions of new software development and deployment techniques. Unless proactive investment in acquisition research and development can give rise to worked examples, open source models, and new acquisition management system technologies, the likelihood of acquisition management dominating agile development and adaptive deployment of component-based OA C2 system capabilities.

Overall, this report serves to help describe and detail how Web-based and mobile device software component technologies, IP licenses, security requirements, business models, and adaptive system evolution interact. It also highlights what policies, practices, or technologies within the DoD and other government agencies can simplify or exacerbate OA system cost arising at different points in the acquisition life cycle. Our common goal is to increase the ways, means, and beneficial consequences of the transition to the costeffective acquisition of Web-based and mobile device OA software systems whose acquisition, development, deployment, and ongoing evolution are agile and adaptive.

References

[AAS09] Alspaugh, T.A, Asuncion, H. and Scacchi, W. (2009). Intellectual property rights requirements for heterogeneously-licensed systems. In *Proc. 17th IEEE International Requirements Engineering Conference (RE'09)*, 24–33, Aug. 31–Sept. 4 2009.

[AAS13] Alspaugh, T.A, Asuncion, H. and Scacchi, W. (2013). The Challenge of Heterogeneously Licensed Systems in Open Architecture Software Ecosystems, in S. Jansen, S. Brinkkemper, and M. Cusumano (Eds.), *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry,* Edward Elgar Publishing, 103-120, Northampton, MA.

[AlS12] Alspaugh, T.A, Scacchi, W. (2012). Security Licensing, *Proc. Fifth Intern. Workshop on Requirements Engineering and Law*, 25-28, September 2012.

[ASA10] Alspaugh, T.A, Scacchi, W., and Asuncion, H. (2010). Software Licenses in Context: The Challenge of Heterogeneously Licensed Systems, *Journal of the Association for Information Systems*, 11(11), 730-755, November 2010.

[And12] Anderson, S. (2012). Software Licensing – Smart Spending in These Changing Times, *CHIPS: The Department of the Navy's Information Technology Magazine*, July-September, 28-31.

[BCK03] Bass, L., Clements, P., and Kazman, R., (2003). *Software Architecture in Practice*, 2nd Edition, Addison-Wesley Professional,New York..

[BeJ10] Bergey, J., & Jones, L. (2010). Exploring acquisition strategies for adopting a software product line approach. *Proc. 7th Acquisition Research Symposium*. Vol. 1, 111-122, Naval Postgraduate School, Monterey, CA.

[Bos00] Bosch, J. (2000). *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley Professional, New York.

[BrA05] Breaux, T.D. and Anton, A.I. (2005). Analyzing goal semantics for rights, permissions, and obligations. In *Proc. 13th IEEE International Conference on Requirements Engineering (RE'05)*, 177–188, 2005.

[BrA08] Breaux, T.D. and Anton, A.I. (2008). Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering*, 34(1), 5–20, 2008.

[ChS90] Choi, S. and Scacchi, W. (1990). Extracting and Restructuring the Design of Large Systems, *IEEE Software*, 7(1), 66-71.

[ChS01] Choi, S.J. And Scacchi, W. (2001). Modeling and Simulating Software Acquisition Process Architectures, *Journal of Systems and Software*, 59, 343-354, 2001.

[ClN01] Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, New York, 2001.

[CrS02] Crowston, K. and Scozzi, B., (2002). Open Source Software Projects as Virtual Organizations. *IEE Proceedings--Software*, 149, 1, 3-17.

[DAU12], Defense Acquisition University (2012). *Open Systems Architecture and Technical Data Rights...Management Approaches*, http://bbp.dau.mil/docs/Open %20Systems%20Architecture%20and%20Technical%20Data%20Rights %20.%20.%20.%20Management%20Approaches.pdf accessed 30 October 2012.

[DIS12], Defense Information Systems Agency (2012). *DOD Open Source and Community Source Software Development in Forge.mil*, SoftwareForge Document ID – doc26066doc26066 http://www.disa.mil/News/Conferences-and-Events/DISA- Mission-Partner-Conference-2012/~/media/Files/DISA/News/Conference/2012/ DoD_Open_Source_Community_Forge.pdf accessed 30 October 2012.

[End13] Endres-Niggemeyer, B. (2013). The Mashup Ecosystem, in *Semantic Mashups: Intelligence Reuse of Web Resources*, Springer, 1-50.

[End13a] Endres-Niggemeyer, B. (2013). Mashups Live on Standards, in *Semantic Mashups: Intelligence Reuse of Web Resources*, Springer, 51-89.

[FIR04] D. Firesmith. Specifying reusable security requirements. *Journal of Object Technology*, 3(1), 61-75, Jan-Feb. 2004.

[GBC14] George, A., Bowers, A., Galdorisi, G., Hszieh, S., Morris, M., and Raney, C. (2014). DoD Application Store: Enabling C2 Agility, *Proc. 19Th Intern. Command and Control Research and Technology Symposium*, Paper-104, Alexandria, VA, June 2014.

[GMH13] George, A., Morris, M., Galdorisi, G., Raney, C., Bowers, A., and Yetman, C. (2013) Mission Composable C3 in DIL Information Environments using Widgets and App Stores. *Proc.*

18Th Intern. Command and Control Research and Technology Symposium, Paper-036, Alexandria, VA, June 2013.

[GuC10] Guertin, N. and Clements, P. (2010). Comparing Acquisition Strategies: Open Architecture versus Product Lines, Vol. 1, 78-90, *Proc. 7th Acquisition Research Symposium*, Naval Postgraduate School, Monterey, CA.

[GuW12] Guertin, N. and Womble, B. (2012). Competition and the DoD Marketplace, *Proc. 9th Acquisition Research Symposium*. Vol. 1, 76-82, Naval Postgraduate School, Monterey, CA.

[JBA08] Jansen, A., Bosch, J., and Avgeriou, P. (2008). Documenting After the Fact: Recovering Architectural Design Decisions, *J. Systems and Software*, 81(4), 536-557.

[JeS05] Jensen, C. and Scacchi, W. (2005). Process Modeling Across the Web Information Infrastructure, *Software Process--Improvement and Practice,* 10(3), 255-272, July-September 2005.

[JoB11] Jones, L. and Bergey, J. (2011). An Architecture-Centric Approach for Acquiring Software-Reliant System, *Proc. 8th Acquisition Research Symposium*, Vol. 1, Naval Postgraduate School, Monterey, CA.

[KaC99] Kazman, R. and Carrière, J. (1999). Playing Detective: Reconstructing Software Architecture from Available Evidence. *J. Automated Software Engineering*, 6(2), 107-138.

[Ke12] Kenyon, H. (2012). DoD, Intel Officials Bullish On Open Source Software; Government-wide Software Foundation In The Mix, *AOL Defense,* October 2012.

[Kon09] Konary, A. (2009). *Software Licensing and Entitlement Management: The Next Software Licensing and Entitlement Management: The Next Generation,* IDC White Paper, October 2009.
http://learn.flexerasoftware.com/content/ECM-WP-Software-Licensing-Entitlement-Management , accessed 2 February 2013.

[MRT99] Medvidovic, N.,Rosenblum, D.S., and Taylor, R.N. (1999). A Language and Environment for Architecture-Based Software Development and Evolution. In Proc. 21st Intern. Conf. Software Engineering (ICSE '99). 44-53, IEEE Computer Society. Los Angeles, CA.

[Nis98] Nissen, M.E. (1998). Redesigning Reengineering through Measurement-Driven Inference, *MIS Quarterly*, 22(4). 509-534.

[NoS99] Noll, J. and Scacchi, W., (1999). Supporting Software Development in Virtual Enterprises, *Jour. Digital Information*, 1(4), February.

[RBC12] Reed, H., Benito, P., Collens, J., and Stein, F. (2012). Supporting Agile C2 with an Agile and Adaptive IT Ecosystem, *17th. Intern. Command and Control Research and Technology Symposium* (ICCRTS), Paper-044, Fairfax, VA, June 2012.

[Sca01] Scacchi, W. (2001). Redesigning Contracted Services Procurement for Internet-Based Electronic Commerce: A Case Study, *J. Information Technology and Management*, 2(3), 313-334.

[Sca07] Scacchi, W., (2007). Free/Open Source Software Development: Recent Research Results and Methods, in M. Zelkowitz (Ed.), *Advances in Computers*, 69, 243-295.

[ScA08] Scacchi, W. and Alspaugh, T., (2008). Emerging Issues in the Acquisition of Open Source Software within the U.S. Department of Defense, *Proc. 5th Acquisition Research Symposium*, NPS-AM-08-036, Naval Postgraduate School, Monterey, CA, May.

[ScA11] Scacchi, W. and Alspaugh, T., (2011). Advances in the Acquisition of Secure Systems Based on Open Architectures, *Proc. 8th Acquisition Research Symposium*, Vol. 1, Naval Postgraduate School, Monterey, CA.

[ScA12a] Scacchi, W. and Alspaugh, T., (2012a) Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems, *Journal of Systems and Software*, 85(7), 1479-1494, July 2012.

[ScA12b] Scacchi, W. and Alspaugh, T., (2012b). Addressing Challenges in the Acquisition of Secure Software Systems with Open Architectures, *Proc. 9th Acquisition Research Symposium*, Vol. 1, 165-184, Naval Postgraduate School, Monterey, CA.

[ScA13a] Scacchi, W. and Alspaugh, T. (2013a). Processes in Securing Open Architecture Software Systems, *Proc. 2013 Intern. Conf. Software and System Processes*, San Francisco, CA, May 2013.

[ScA13b] Scacchi, W. and Alspaugh, T. (2013b). Challenges in the Development and Evolution of Secure Open Architecture Command and Control Systems, *Proc. 18th. Intern. Command and Control Research and Technology Symposium*, Paper-098, Alexandria, VA, June 2013.

[ScB12] Scacchi, W., Brown, C. and Nies, K. (2012). Exploring the Potential of Virtual Worlds for Decentralized Command and Control, *Proc. 17th. Intern. Command and Control Research and Technology Symposium* (ICCRTS), Paper 096, Fairfax, VA, June 2012.

[SFF06] Scacchi, W., Feller, J., B. Fitzgerald, Hissam, S.and Lakhani, K., (2006). Understanding Free/Open Source Software Development Processes, *Software Process--Improvement and Practice*, 11(2), 95-105, March/April.

[ScJ08] Scacchi, W. and Jensen, C., (2008). Governance in Open Source Software Development Projects: Towards a Model for Network-Centric Edge Organizations, *Proc. 13th. Intern. Command and Control Research and Technology Symp.*, Bellevue, WA, (to appear, June).

[ScN97] Scacchi, W. and Noll. J. (1997). Process-Driven Intranets: Life Cycle Support for Process Reengineering, *IEEE Internet Computing*, 1(5):42-49.

[SEI07] Northrop, L., & Clements, et al., Software Engineering Institute (2007). *A Framework for Software Product Line Practice*, Version 5.0.
http://www.sei.cmu.edu/productlines/ frame_report/index.html

[WoS11] Womble, B., Schmidt, W., Arendt, M., and Fain, T. (2011). Delivering Savings with Open Architecture and Product Lines, *Proc. 8th Acquisition Research Symposium*, Vol. 1, Naval Postgraduate School, Monterey, CA.

[YaC06] S. S. Yau and Z. Chen. A framework for specifying and managing security requirements in collaborative systems. In *Proc. Third International Conference on Autonomic and Trusted Computing (ATC 2006)*, 500–510, 2006.