UCI-AM-11-203

ACQUISITION RESEARCH SPONSORED REPORT SERIES

Investigating Advances in the Acquisition of Secure Systems Based on Open Architecture, Open Source Software, and Software Product Lines

27 January 2012

by

Dr. Walt Scacchi, Senior Research Scientist, and

Dr. Thomas A. Alspaugh, Assistant Professor

Institute for Software Research

University of California, Irvine

Prepared for: Naval Postgraduate School, Monterey, California 93943

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

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

To request Defense Acquisition Research or to become a research sponsor, please contact:

NPS Acquisition Research Program
Attn: James B. Greene, RADM, USN, (Ret)
Acquisition Chair
Graduate School of Business and Public Policy
Naval Postgraduate School
555 Dyer Road, Room 332
Monterey, CA 93943-5103
Tel: (831) 656-2092
Fax: (831) 656-2253
e-mail: jbgreene@nps.edu

Copies of the Acquisition Sponsored Research Reports may be printed from our website www.acquisitionresearch.org

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

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

I. Advances in the Acquisition of Secure Systems Based on Open Architectures

Walt Scacchi and Thomas A. Alspaugh
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
wscacchi@ics.uci.edu, thomas.alspaugh@acm.org

Abstract

The role of software acquisition ecosystems in the development and evolution of secure open architecture systems has received insufficient consideration. Such systems are composed of software components subject to different security requirements in an architecture in which evolution can occur by evolving existing components or by replacing them. However, this may result in possible security requirements conflicts and organizational liability for failure to fulfill security obligations. We have developed an approach for understanding and modeling software security requirements as "security licenses," for analyzing conflicts among groups of such licenses in realistic system contexts, and for guiding the acquisition, integration, or development of systems with open source components in such an environment. Consequently, this paper reports on our efforts to extend our existing approach to specifying and analyzing software intellectual property licenses to now address software security licenses that can be associated with secure OA systems.

Biographies

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 1981–1998, 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 45 externally funded research projects. In 2007,

he served as General Chair of the 3rd IFIP International Conference on Open Source Systems (OSS2007), Limerick, IE. In 2010, he chaired the Workshop on the Future of Research in Free and Open Source Software, Newport Beach, CA, for the Computing Community Consortium and the National Science Foundation. He also serves as Co-Chair of the Software Engineering in Practice (SEIP) Track at the 33rd International Conference on Software Engineering, May 21–28, 2011, Honolulu, HI.

Thomas Alspaugh is adjunct professor of Computer Science at Georgetown University and a visiting researcher at the Institute for Software Research at UC Irvine. His research interests are in software engineering and software requirements. Before completing his Ph.D., 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 project, also known as the A-7E project.

A. Introduction

A substantial number of development organizations are adopting a strategy in which a software-intensive system is developed with an open architecture (OA; Oreizy, 2000), whose components may be open source software (OSS) or proprietary with open application programming interfaces (APIs). Such systems evolve not only through the evolution of their individual components, but also through replacement of one component by another, possibly from a different producer or under a different license. With this approach to software system acquisition, the system development 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 arises in a software acquisition 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 subject to different security requirements. This may arise due either to how a component's external interfaces are specified and defended, or to how system components are interconnected and configured in ways that can or cannot defend the composed system from security vulnerabilities and external exploits. Ideally, any software element in a system composed from components from different producers can have its security capabilities specified, analyzed, and implemented at system architectural design-time, build-time, or at deployment run-time. Such capability-based security in its simplest form specifies what types, value ranges, and values of data, or control signals (e.g., program invocations, procedure or method calls), can be input, output, or handed off to a software plug-in or external (helper) application from a software component or composed system.

When designing a secure OA system, decisions and trade-offs must be made as to what level of security is required, as well as to what kinds of threats to security must be addressed. The universe of possible security threats is continually emerging and the cost/effort of defending against them is ongoing. Similarly, anticipating all possible security vulnerabilities or threats is impractical (or impossible). Further, though it may be desirable that all systems be secure, different systems need different levels of security, which may come at ever greater cost or inconvenience in order to accommodate. Strategic systems may need the greatest security possible, whereas other systems may require much less rigorous security mechanisms. Thus, finding an affordable, scalable, and testable means for specifying the security requirements of software components, or OA systems composed with components with different security requirements, is the goal of our research.

The most basic form of security requirements that can be asserted and tested are those associated with virtual machines. Virtual machines (VM) abstract away the

actual functional or processing capabilities of the computational systems on which they operate, and instead provide a limited functionality computing surround (or "sandbox"). VM can isolate a given component or system, other software applications, utilities, repositories, or external/remote control data access (input or output). The capabilities for a VM (e.g., an explicit, pre-defined list of approved operating system commands or programs that can write data or access a repository) can be specified as testable conditions that can be assigned to users or programs authorized to operate within the VM. The VM technique is now widely employed through software "hypervisors" (e.g., IBM VM/370, VMware, VirtualBox, Parallels Desktop for Mac) that isolate software applications and operating systems from the underlying system platform or hardware. Such VM act like "containment vessels" through which it is possible to specify barriers to entry (and exit) of data and control via security capabilities that restrict other programs. Thus, these capabilities specify what rights or obligations may be, or may not be, available for access or update to data or control information. Thus, architectural design-time decisions pertaining to specifying the security rights or obligations for the overall system or its components are done by specification of VM that contain the composed system or its components. These rights or obligations can be specified as pre-conditions on input data or control signals, or post-conditions on output data or control signals.

The problem of specifying the build-time and run-time security requirements of OA systems is different from that at design-time. In determining how to specify the software build sequence, security requirements are manifest as capabilities that may be specific to explicitly declared versions of designated programs. For example, if an OA system at design-time specifies a "Web browser" as one of its components, at build-time a particular Web browser (Mozilla Firefox or Internet Explorer) must then be specified, as must its baseline version (e.g., Firefox 4.0 or Internet Explorer 9.0). However, if the resulting run-time version of the OA system must instead employ a locally available Web browser (e.g., Firefox 3.6.1 or Internet Explorer 8.0 Service Pack 2), then the OA system integrators may either need to produce multiple run-time versions for deployment, or else build the OA system using (a) an earlier

version of the necessary component (e.g., Firefox 3.5 or Internet Explorer 7.0) that is "upward compatible" with newer browser versions, (b) a stub or abstract program interface that allows for a later designated compatible component version to be installed/used at run-time, or else (c) create different run-time version alternatives (i.e., variants) of the target OA systems that may or may not be "backward compatible" with the legacy system component versions available in the deployment run-time environment. The need to specify build-time and run-time components by hierarchical version numbers like Firefox 3.6.16.144 (and possibly timestamps of their creation or local installation) arises because evolutionary version updates often include security patches that close known vulnerabilities or prevent known exploits. As indicated in Section 2, Related Work, security attacks often rely on system entry through known vulnerabilities that are present in earlier versions of software components that have not been updated to newer versions that don't have the same vulnerabilities.

Because we have been able to address an analogous problem of how to specify and analyze the intellectual property rights and obligations of the licenses of software components, our efforts now focus on the challenge of how to specify and analyze software components and composed system security rights and obligations using a new information structure we call a "security license." The actual form of such a security license is still to be finalized, but at this point, we believe it is appropriate to begin to develop candidate forms or types of security licenses for further research and development, especially for security license forms that can be easily formalized, readily applied to large-scale OA systems, and be automatically analyzed or tested in ways that we have already established (Alspaugh, Asuncion, & Scacchi, 2010; Alspaugh, Scacchi, & Asuncion, 2010). This is another goal of our research here.

Next, the challenge of specifying secure software systems composed from secure or insecure components is inevitably entwined with the software ecosystems that arise for secure OA systems. We found that an OA software acquisition

ecosystem involves organizations and individuals producing and consuming components, and supply paths from producer to consumer; but also

- the OA of the system(s) in question, and how best to secure it,
- the open interfaces provided by the components, and how to specify their security requirements,
- the degree of coupling in the evolution of related components that can be assessed in terms of how security rights and obligations may change, and
- the rights and obligations resulting from the security licenses under which various components are released that propagate from producers to consumers.

An example of a software acquisition ecosystem producing and integrating secure software components or secure systems is portrayed in Figure 1.

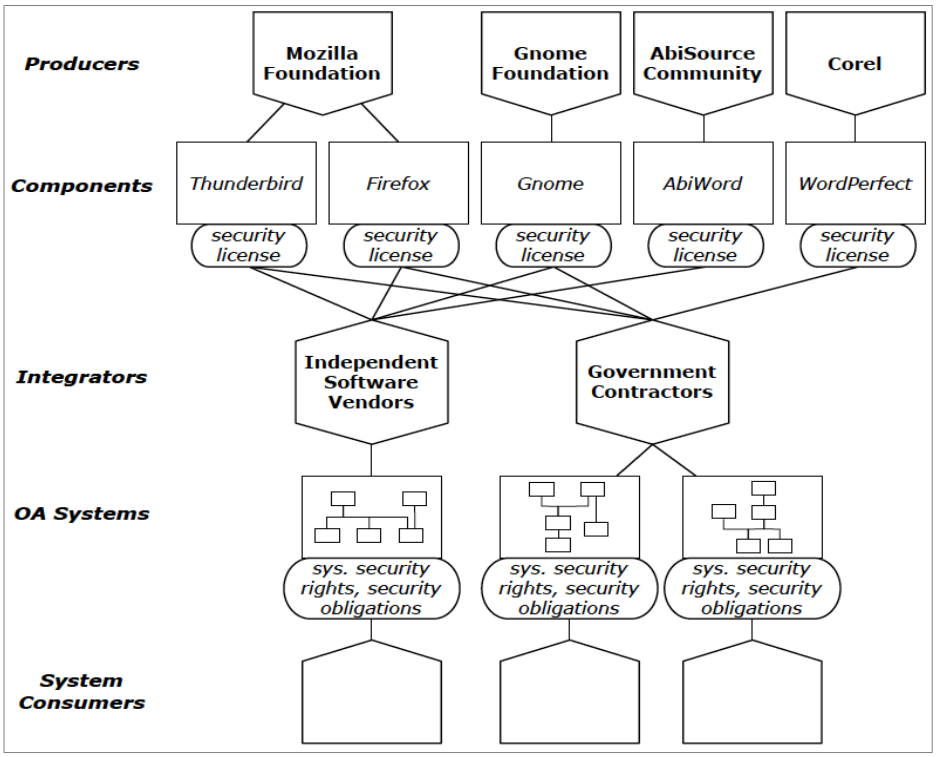

Figure 1. An Example of a Software Acquisition Ecosystem in Which Secure OA Systems May Be Developed

In order to most effectively use an OA approach in developing and evolving a system, it is essential to consider this OA ecosystem. An OA system draws on components from proprietary vendors and open source projects. Its architecture is made possible by the existing general ecosystem of producers, from which the initial components are chosen. The choice of a specific OA begins a specialized software ecosystem involving components that meet (or can be shimmed to meet) the open interfaces used in the architecture. We do not claim that this is the best or the only way to reuse components or to produce secure OA systems, but it is an ever more widespread way. In this paper we built on previous work on heterogeneously licensed systems (Alspaugh, Asuncion, et al., 2009a; German & Hassan, 2009; Scacchi & Alspaugh, 2008) by examining how OA development affects and is affected by software ecosystems and the role of security licenses for components included within OA software ecosystems.

In the remainder of this paper, we survey some related work (Section 2), define and examine characteristics of open architectures with or without secure software elements (Section 3), define and examine characteristics for how secure OA systems evolve (Section 4), introduce a structure for security licenses (Section 5), outline security license architectures (Section 6), and sketch our approach for security license analysis (Section 7). We then close with a discussion addressing how our software license and analysis scheme relates to software product lines (Section 8) before stating our conclusions (Section 9).

B. Related Work

Software systems, whether operating as standalone components or as elements within large system compositions, are continuously being subjected to security attacks. These attacks seek to slip through software vulnerabilities known to the attackers but perhaps not to the system integrators or consumers. These attacks often seek to access, manipulate, or remotely affect the data values or control signals that a component or composed system processes for nefarious purposes or seek to congest or over-saturate networked services. Recent high profile security

attacks like *Stuxnet* (Falliere, Murchu, & Chien, 2011) reveal that security attacks may be very well planned and employ a bundle of attack vectors and social engineering tactics in order for the attack to reach strategic systems that are mostly isolated and walled off from public computer networks. The Stuxnet attack entered through software system interfaces at either the component, application subsystem, or base operating system level (e.g., via removable thumb drive storage devices), and their goal was to go outside or beneath their entry context. However, all of the Stuxnet attacks on the targeted software system could be blocked or prevented through security capabilities associated with the open software interfaces that would (a) limit access or evolutionary update rights lacking proper authorization, as well as through (b) "sandboxing" (i.e., isolating) and holding up any evolutionary updates (the attacks) prior to their installation and run-time deployment. Furthermore, because the Stuxnet attack involved the use of corrupted certificates of trust from approved authorities as false credentials that allowed evolutionary system updates to go forward, it seems clear that additional preventions are needed that are external to, and prior to, their installation and run-time deployment. In our case, that means that we need to specify and analyze software security requirements and evolutionary update capabilities at architectural design-time and system integration build-time, and then reconcile those with the run-time system composition. It also calls for the need to maintain the design-time, build-time, and run-time system compositions in repositories remote from system installations, and in possibly redundant locations that can be encrypted, randomized, fragmented, and dispersed (e.g., via Torrents or "onion routing") then cross-checked and independently verified prior to run-time deployment in a high security system application.

As already noted, both software intellectual property licenses and security 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 system or system components as intellectual property (IP) or security requirements (i.e., capabilities) during their development and deployment. However, 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 (Breaux & Anton, 2005, 2008). 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 (Yau & Chen, 2006). Using a semantic model to formally specify the rights and obligations required for a software system or component to be secure (Breaux & Anton, 2005, 2008; Yau & Chen, 2006) means that it may be possible to develop both a "security architecture" notation and model specification that associates given security rights and obligations across a software system or system of systems. Similarly, it suggests the possibility of developing computational tools or interactive architecture development environments that can be used to specify, model, and analyze a software system's security architecture at different times in its development—design-time, build-time, and run-time. The approach we have been developing for the past few years for modeling and analyzing software system IP license architectures for OA systems (Alspaugh, Asuncion, et al., 2009b, 2010; Aslpaugh, Scacchi, et al., 2010; Scacchi & Alspaugh, 2008) 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 (2004) at the Software Engineering Institute. But 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.

C. Secure Open Architecture Composition

Open architecture (OA) software is a customization technique introduced by Oreizy (2000) that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. Increasingly more software-intensive systems are developed using an OA strategy, not only with open source software (OSS) components but also proprietary components with open APIs. Similarly, these components may or may not have their own security requirements that must be satisfied during their build-time integration or run-time deployment, such as registering the software component for automatic update and installation of new software versions that patch recently discovered security vulnerabilities or prevent invocation of known exploits. Using this approach can lower development costs and increase reliability and function as well as adaptively evolve software security (Scacchi & Alspaugh, 2008). Composing a system with heterogeneously secured components, however, increases the likelihood of conflicts, liabilities, and no-rights stemming from incompatible security requirements. Thus, in our work we define a secure 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, such that they do not introduce new security vulnerabilities at the system architectural level*.

It may appear that using a system architecture that incorporates secure OSS and proprietary components and uses open APIs will result in a secure OA system. However, not all such architectures will produce a secure OA because the (possibly empty) set of available license rights for an OA system depends on (a) how and why secure or insecure components and open APIs are located within the system architecture, (b) how components and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the IP and security licenses of different OSS components encumber all or part of a software system's architecture into which they are integrated (Alspaugh & Anton, 2008; Scacchi & Alspaugh, 2008).

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

Software source code components—These can be either (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code such as C shell scripts, (d) intra-application script code, such as for creating Rich Internet Applications using domain-specific languages such as XUL for the Firefox Web browser (Feldt, 2007) or "mashups" (Nelson & Churchill, 2006), whose source code is available and they can be rebuilt, or (e) similar script code that can either install and invoke externally developed plug-in software components or invoke external application (helper) components. Each may have its own distinct IP/security requirements.

Executable components—These components are in binary form and the source code may not be open for access, review, modification, or possible redistribution (Rosen, 2005). If proprietary, they often cannot be redistributed, and so such components will be present in the design- and run-time architectures but not in the distribution-time architecture.

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

Application programming interfaces/APIs—Availability of externally visible and accessible APIs is the minimum requirement for an "open system" (Meyers & Oberndorf, 2001).

Software connectors—The intended purpose of this software is to provide a standard or reusable way of communication through common interfaces (e.g., High Level Architecture [Kul, Weatherly, & Dahmann, 1999], CORBA, MS .NET, Enterprise Java Beans, and GNU Lesser General Public License [LGPL] libraries).

Connectors can also limit the propagation of IP license obligations or provide additional security capabilities.

Methods of connection—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. Methods of connection affect license obligation propagation, with different methods affecting different licenses.

Configured system or subsystem architectures—These are software systems that are used as atomic components of a larger system and whose internal architecture may comprise components with different licenses, affecting the overall system license and its security requirements. To minimize license interaction, a configured system or sub-architecture may be surrounded by what we term a *license firewall,* namely a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of reciprocal obligations.

Figure 2 shows a high-level, run-time view of a composed OA system whose reference architectural design in Figure 3 includes all of the kinds of software elements listed in the previous paragraphs. This reference architecture has been instantiated in a build-time configuration in Figure 4 that in turn could be realized in alternative run-time configurations in Figures 5, 6, and 7 with different security capabilities. The configured systems consist of software components such as a Mozilla 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. The components are interconnected through a set of software connectors that bridge the interfaces of components and combine the provided functionality into the system's services. However, note how the run-time software architecture does not pre-determine how security capabilities will be assigned and distributed across different variants of the run-time composition.

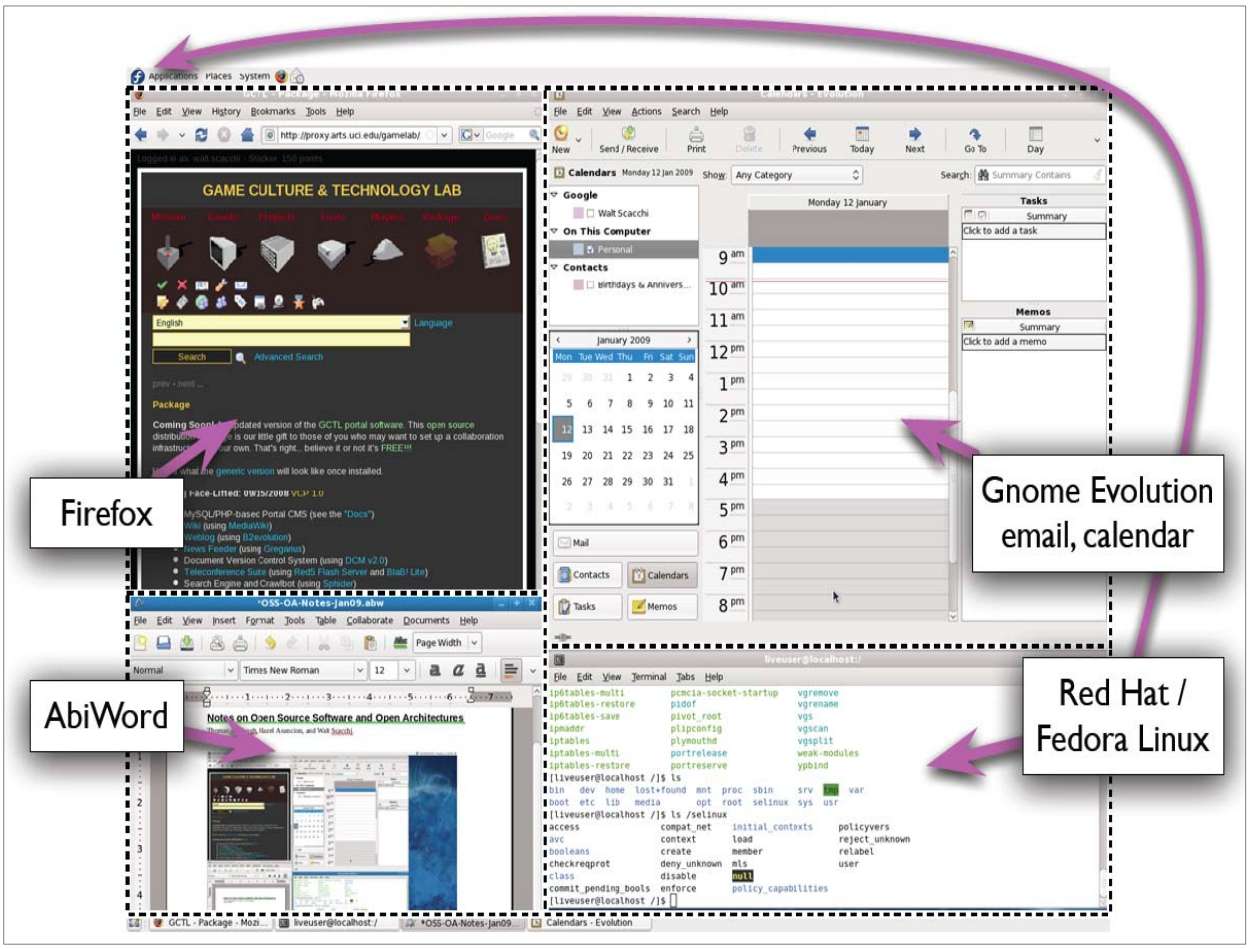

Figure 2. An Example Composite OA System Potentially Subject to Different IP and Security Licenses

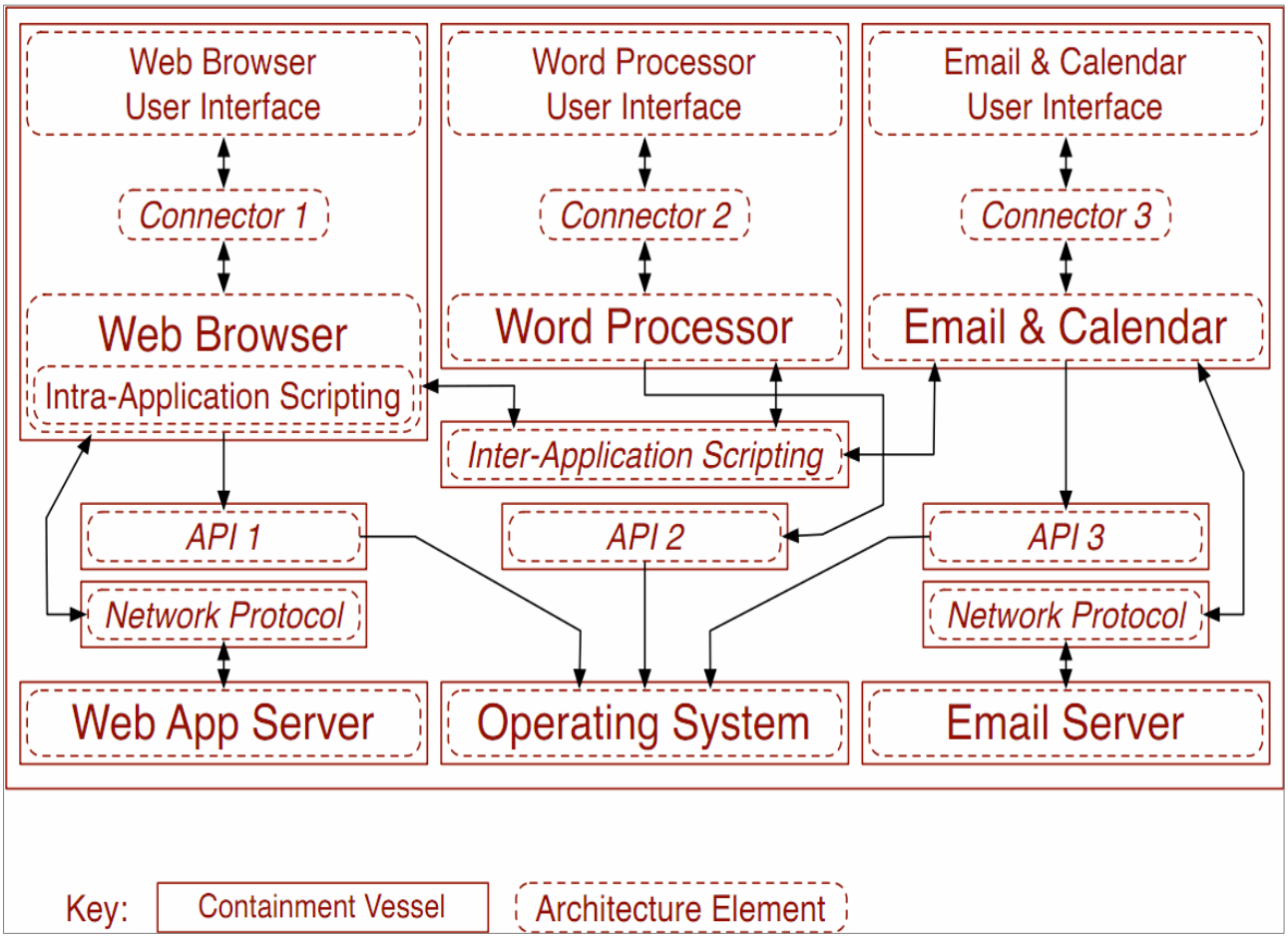

Figure 3. The Design-Time Architecture of the System in Figure 2 That Specifies a Required Security Containment Vessel Scheme

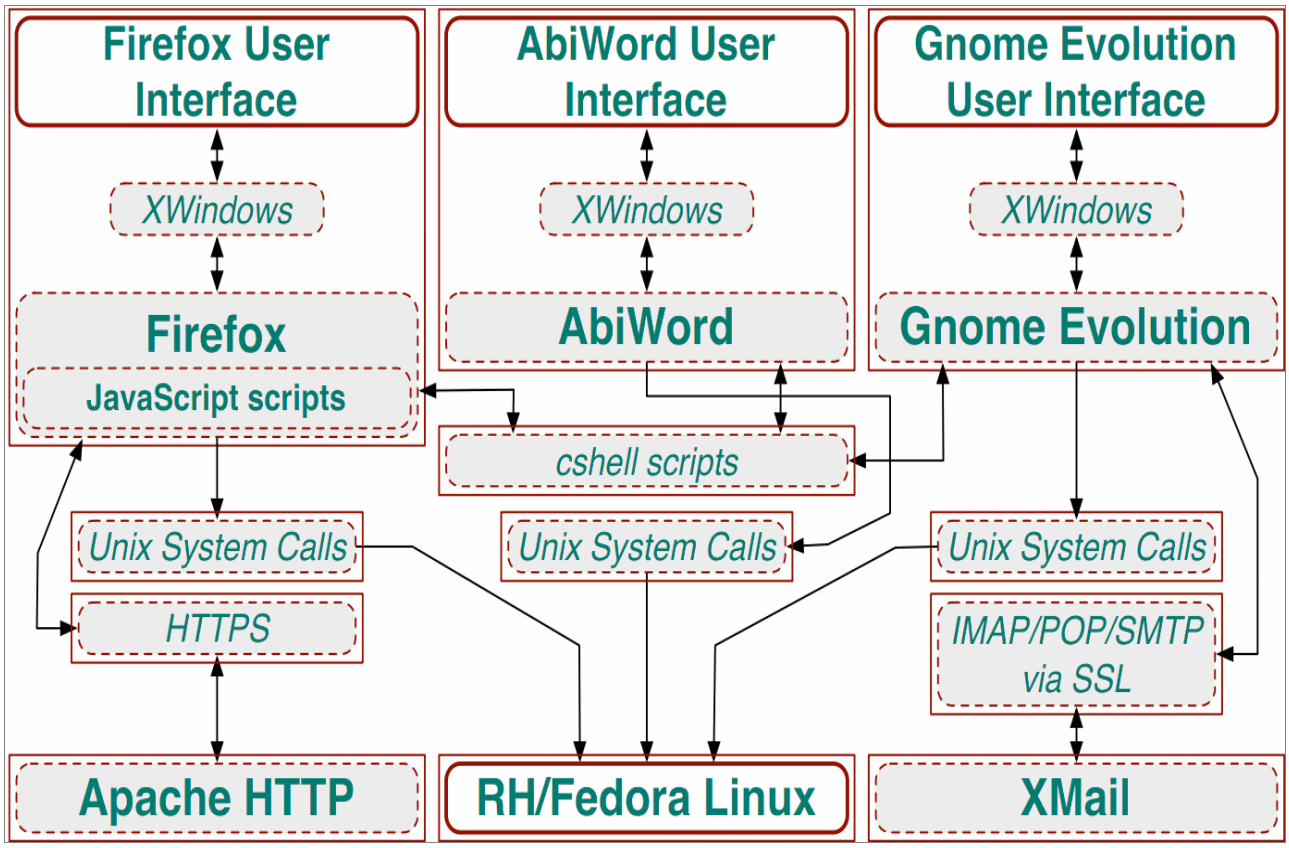

Figure 4. A Secure Build-Time Architecture Describing the Version Running in Figure 2 with a Specified Security Containment Vessel Scheme

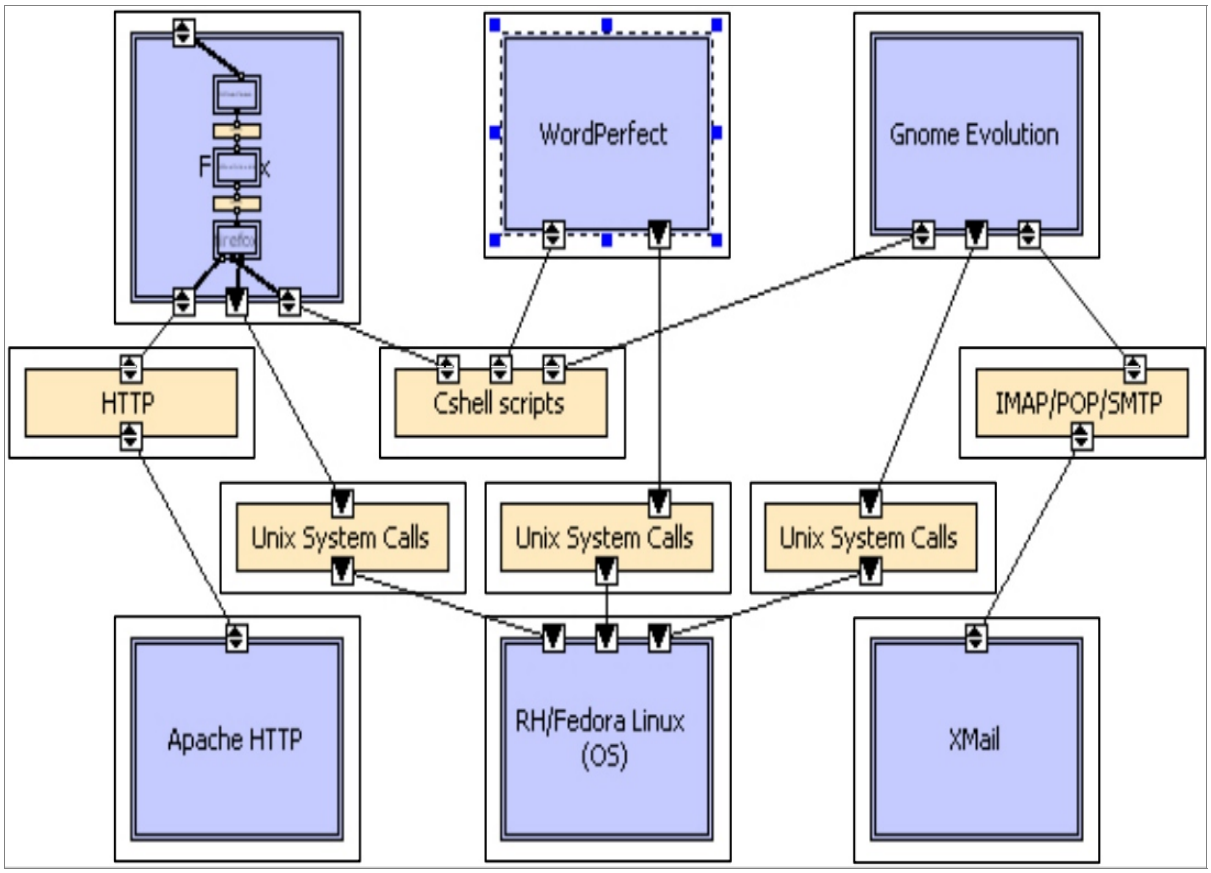

Figure 5. Instantiated Build-Time OA System with Maximum Security Architecture of Figure 4 Via Individual Security Containment Vessels for Each System Element

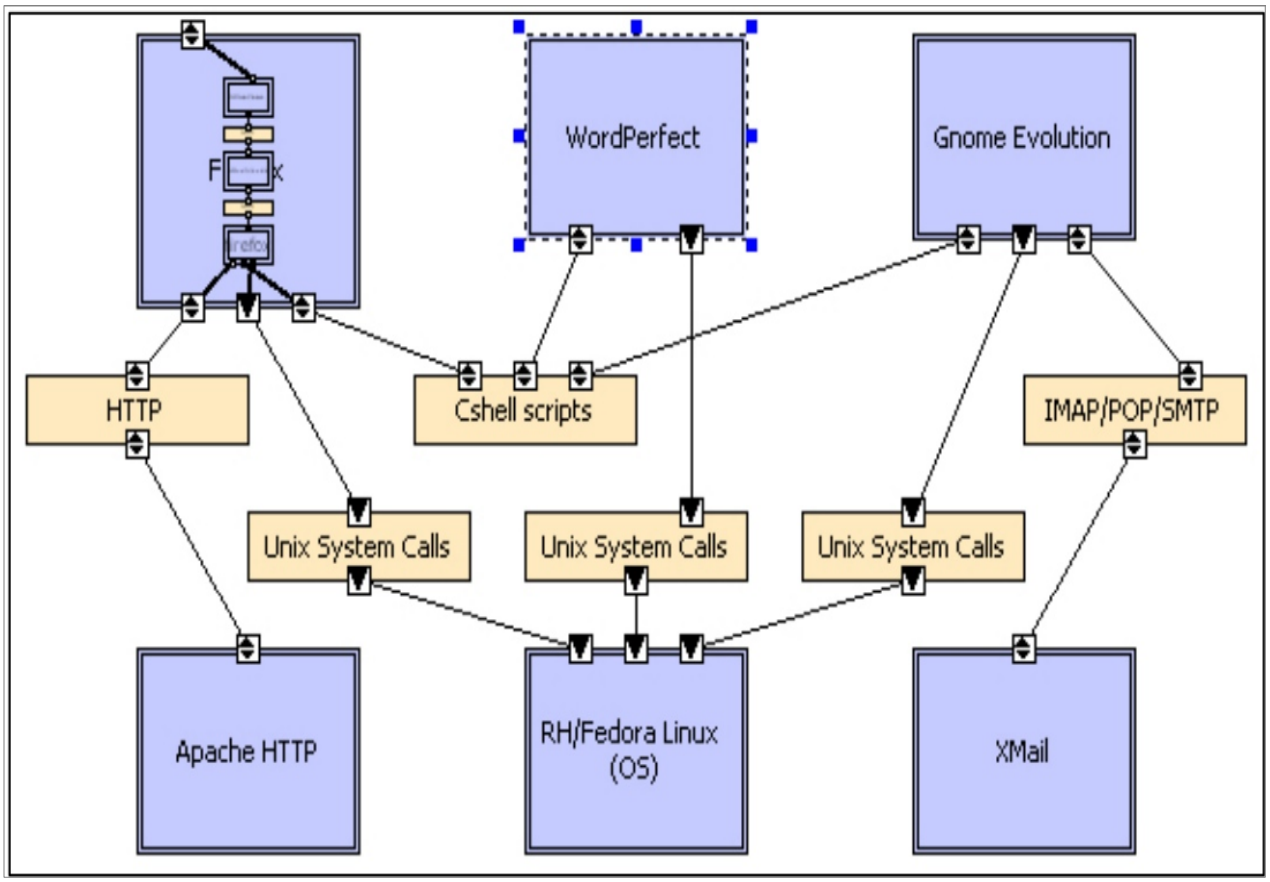

Figure 6. Instantiated Build-Time OA System with Minimum Security Architecture of Figure 4 Via a Single Overall Security Containment Vessel for the Complete System Using a Common Software Hypervisor

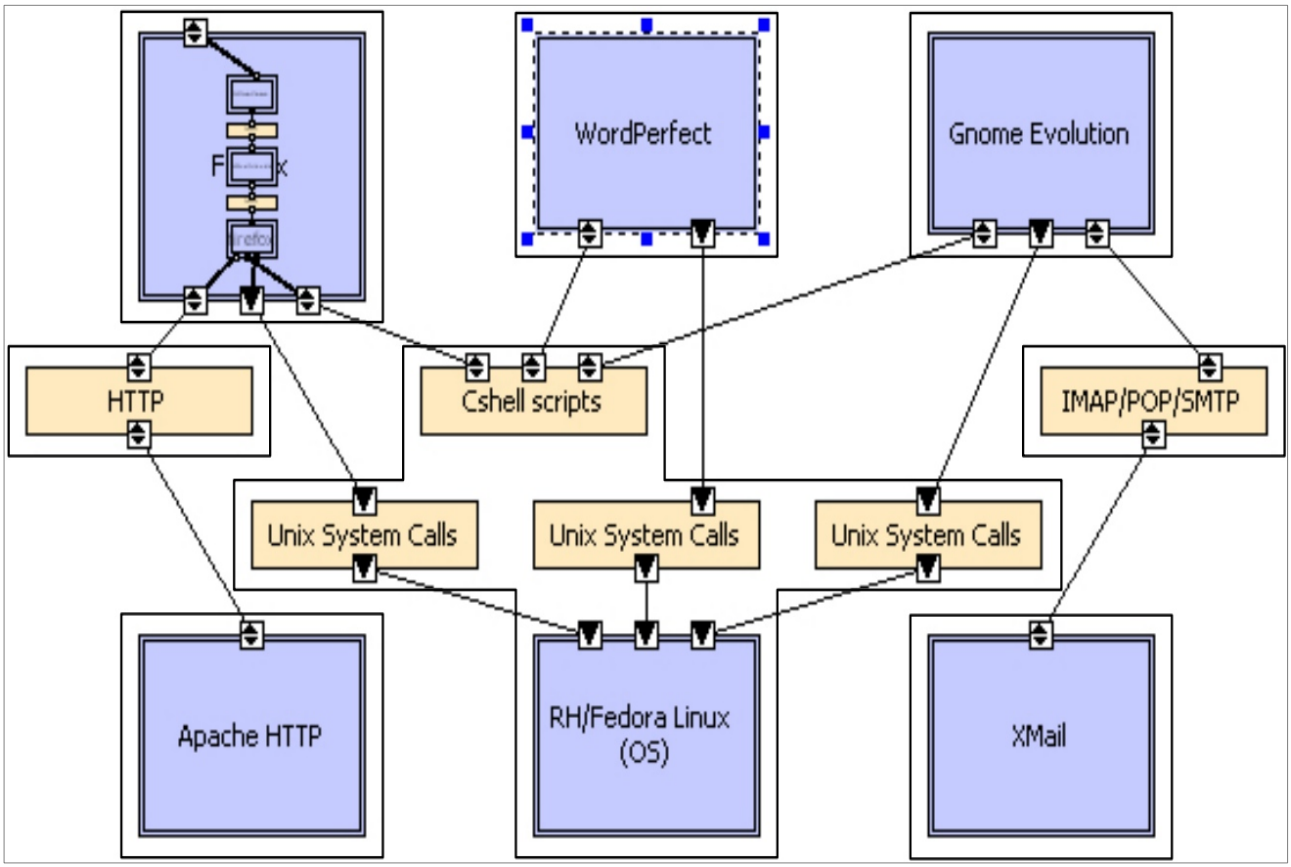

Figure 7. Instantiated Build-Time OA System with Mixed Security Architecture of Figure 4 Via Security Containment Vessels for Some Groupings of System Elements

D. OA System 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 IP and security licenses in a single system.

1. 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 3.5 to 3.6, which may update existing software functionality while also patching recent security vulnerabilities).

2. By component replacement

One or more components may be replaced by others with different behaviors but the same interface, or with a different interface and with the addition of shim code to make it match (for example, replacing the AbiWord word processor with either Open Office or MS Word, depending on which is considered the least vulnerable to security attack).

3. By architecture evolution

The OA can evolve, using the same components but in a different configuration, altering the system's characteristics. For example, as discussed in Section 3, changing the configuration in which a component is connected can change how its IP or security license affects the rights and obligations for the overall system. This could arise when replacing email and word processing applications with web services like Google Mail and Google Docs, which we might assume may be more secure because the Google services (operating in a cloud environment) may not be easily accessed or penetrated by a security attack.

4. 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. Similarly, the security license for a component may be changed by its producers, or the security license for a composed system changed by its integrators, in order to prevent or deter recently discovered security vulnerabilities or exploits before an evolutionary version update (or patch) can be made available.

5. By a change to the desired rights or acceptable obligations

The OA system's integrator or consumers may desire additional IP or security 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 in order 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 that have known vulnerabilities that have not been patched and eliminated.

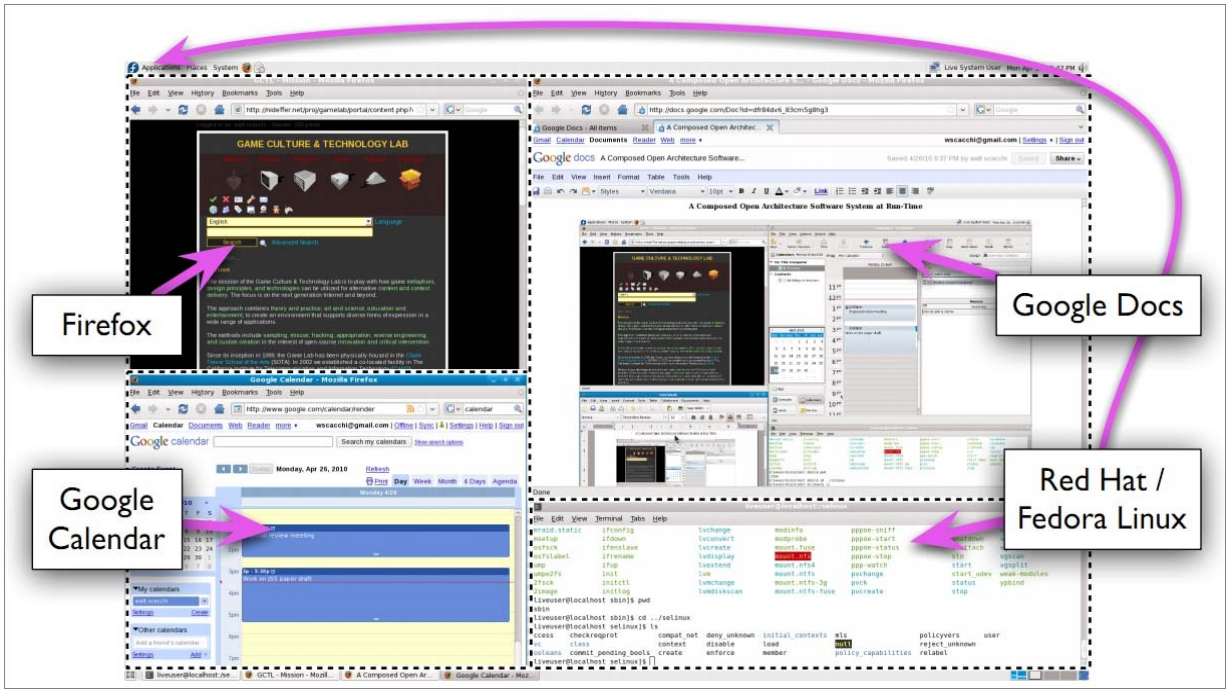

Figure 8. A Second Instantiation at Run-Time

Note. This is the OA system in Figures 2, 3, and 4 (Firefox, Google Docs and Calendar operating within different Firefox run-time sessions, Fedora) as an evolutionary alternative system version, which requires an alternative security containment scheme.

The interdependence of integrators and producers results in a co-evolution of software within an OA ecosystem. Closely-coupled components from different producers must evolve in parallel in order for each to provide its services, as evolution in one will typically require a matching evolution in the other. Producers may manage their evolution with a loose coordination among releases, for example as between the Gnome and Mozilla organizations. Each release of a producer component creates a tension through the ecosystem relationships with consumers and their releases of OA systems using those components, because integrators accommodate the choices of available, supported components with their own goals and needs. As discussed in our previous work (Alspaugh, Asuncion, et al., 2009a), license rights and obligations are manifested at each component's interface, then mediated through the system's OA to entail the rights and corresponding obligations for the system as a whole. As a result, integrators must frequently re-evaluate an OA system's IP/security rights and obligations. In contrast to homogeneously-licensed systems, license change across versions is a characteristic of OA ecosystems, and architects of OA systems require tool support for managing the ongoing licensing changes.

We propose that such support must have several characteristics.

- It must rest on a license structure of rights and obligations (Section 5), focusing on obligations that are enactable and testable.
- It must take account of the distinctions between the design-time, build-time, and distribution-time architectures (Sections 3, 5, 6) and the rights and obligations that come into play for each of them.
- It must distinguish the architectural constructs significant for software licenses, and embody their effects on rights and obligations (Section 3).
- It must define license architectures (Section 6).
- It must provide an automated environment for creating and managing license architectures. We are developing a prototype that manages a license architecture as a view of its system architecture (Alspaugh, Asuncion, et al., 2009a).

- Finally, it must automate calculations on system rights and obligations so that they may be done easily and frequently, whenever any of the factors affecting rights and obligations may have changed (Section 7).

E. Security Licenses

Licenses typically impose obligations that must be met in order for the licensee to realize the assigned rights. Common IP/copyright license obligations include the obligation to publish at no cost any source code you modify (MPL) or the reciprocal obligation to publish all source code included at build-time or statically linked (GPL). The obligations may conflict, as when a GPL'd component's reciprocal obligation to publish source code of other components is combined with a proprietary component's license prohibition of publishing its source code. In this case, no rights may be available for the system as a whole, not even the right of use, because the two obligations cannot simultaneously be met and thus neither component can be used as part of the system. Security capabilities can similarly be expressed and bound to the data values and control signals that are visible in component interfaces, or through component connectors.

Some typical security rights and obligations might be

- the right to read data in containment vessel T,
- the obligation for a specific component to have been vetted for the capability to read and update data in containment vessel T,
- the obligation for a user to verify his/her authority to see containment vessel T by password or other specified authentication process,
- the right to replace specified component C with some other component,
- the right to add or update specified component D in a specified configuration, and
- the right to add, update, or remove a security mechanism.

The basic relationship between software IP/security license rights and obligations can be summarized as follows: if the specified obligations are met, then

the corresponding rights are granted. For example, if you publish your modified source code and sub-licensed derived works under MPL, then you get all of the MPL rights for both the original and the modified code. Similarly, software security requirements are specified as security obligations that when met, allow designated users or other software programs to access, modify, and redistribute data and control information to designated repositories or remote services. However, license details are complex, subtle, and difficult to comprehend and track—it is easy to become confused or make mistakes. The challenge is multiplied when dealing with configured system architectures that compose a large number of components with heterogeneous IP/security licenses, so that the need for legal counsel begins to seem inevitable (Fontana et al., 2008; Rosen, 2005).

We have developed an approach for expressing software licenses of different types (intellectual property and security requirements) that is more formal and less ambiguous than natural language, and that allows us to calculate and identify conflicts arising from the rights and obligations of two or more components' licenses. Our approach is based on Hohfeld's (1913) classic group of eight fundamental jural relations, of which we use right, duty, no-right, and privilege. We start with a tuple <actor, operation, action, object> for expressing a right or obligation. The actor is the "licensee" for all of the licenses we have examined. The operation is one of the following: *may*, *must*, *must not*, or *need not*, with *may* and *need not* expressing rights and *must* and *must not* expressing obligations. The action is a verb or verb phrase describing what may, must, must not, or need not be done, with the object completing the description. A license may be expressed as a set of rights, with each right associated with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 9 shows the meta-model with which we express licenses.

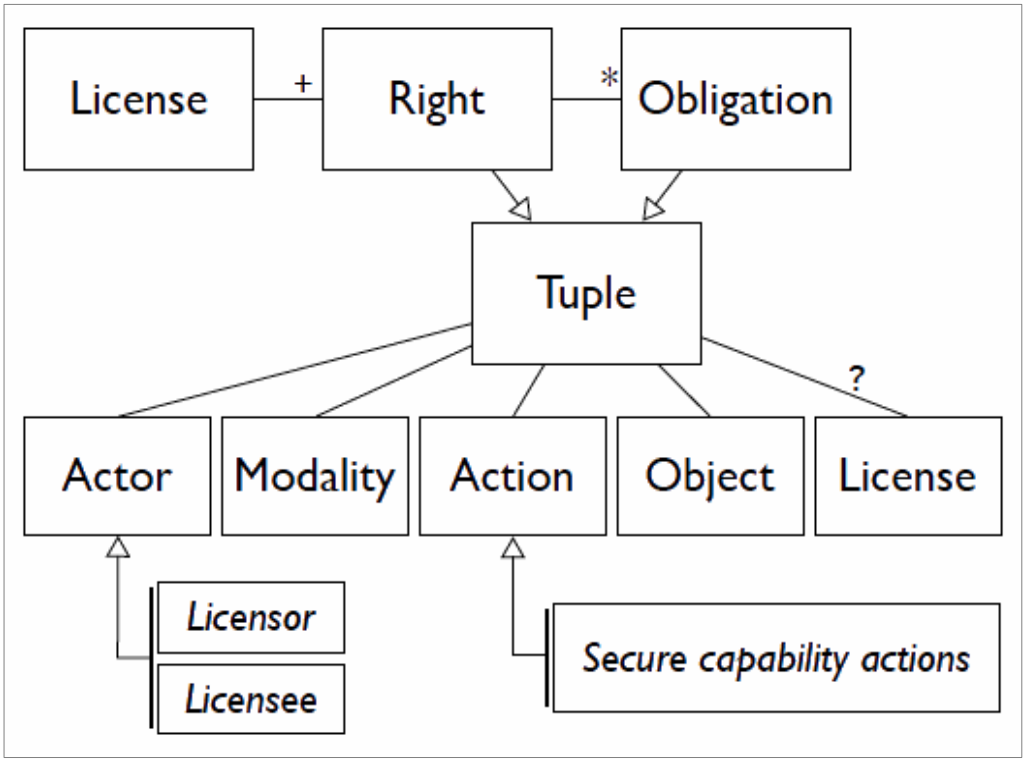

Figure 9. Security License Meta-Model

Designers of secure systems have developed a number heuristics to guide architectural design in order to satisfy overall system security requirements while avoiding conflicts among interacting security mechanisms or defenses. However, even using design heuristics (and there are many), keeping track of security rights and obligations across components that are interconnected in complex OAs quickly becomes too cumbersome. Automated support is needed to manage the complexity of multi-component system compositions where different security requirements must be addressed through different security capabilities.

F. Security License Architectures

Our security license model forms a basis for effective reasoning about licenses in the context of actual systems and for calculating the resulting rights and obligations. In order to do so, we need a certain amount of information about the system's configuration at design-time, build-time, and run-time deployment. The

needed information comprises the license architecture, and an abstraction of the system architecture:

1. the set of components of the system (for example, see Figure 2) for the current system configuration, as well as subsequently for system evolution update versions (as seen in Figure 8);

2. the relation mapping each component to its security requirements (specified and analyzed at design-time, as exemplified in Figure 3) or capabilities (specified and analyzed at build-time in Figure 4 and run-time across alternatives shown in Figure 5, 6, and 7);

3. the connections between components and the security requirements or capabilities of each connector passing data or control signals to/from it; and

4. possibly other information, such as information to detect or prevent IP and security requirements conflicts, which is as yet undetermined.

With this information and definitions of the licenses involved, we believe it is possible to automatically calculate rights and obligations for individual components or for the entire system as well as to guide/assess system design and evolution using an automated environment of the kind that we have previously demonstrated (Alspaugh, Asuncion, et al., 2009a, 2009b, 2010; Alspaugh, Schacchi, et al., 2010).

G. Security License Analysis

Given a specification of a software system's architecture, we can associate security license attributes with the system's components, connectors, and sub-system architectures, resulting in a license architecture for the system, and we can calculate the security rights and obligations for the system's configuration. Due to the complexity of license architecture analysis, and the need to re-analyze every time a component evolves, a component's security license changes, a component is substituted, or the system architecture changes, OA integrators really need an automated license architecture analysis environment. We have developed a prototype of such an environment for analogous calculations for software copyright

licenses (Alspaugh, Asuncion, et al., 2009b; Alspaugh, Schacchi, et al., 2010), and are extending this approach to security licenses.

1. Security Obligation Conflicts

A security obligation can conflict with another obligation, a related right for the same or nearby components, or with the set of available security rights by requiring a right that has not been granted. For instance, consider two connected components C and D with the following security obligations.

> (O1) The obligation for component C to have been vetted for the capability to read and update data in containment vessel T
>
> (O2) The obligation for all components connected to specified component D to grant it the capability to read and update data in containment vessel T

If C has not been vetted, then these two obligations conflict. This possible conflict must be taken into consideration in different ways at different development times:

- at design time, ensuring that it will be possible to vet C;
- at build time, ensuring that the specific implementation of C has been vetted successfully; and
- possibly at run time as well, confirming that C is certified to have been vetted, or (if C is dynamically connected at run time) vetting C before trusting the connection to it.

The second obligation may also conflict with the set of available security rights, for example if D is connected to component E for which the security right

> (R1) to read and update data in containment vessel T using component E is not available.

The absence of such conflicts does not mean, of course, that the system is secure, but the presence of conflicts reliably indicates that it is not secure.

2. Rights and Obligations Calculations

The rights available for the entire system (the right to read and update data in containment vessel T, the right to replace components with other components, the right to update component security licenses, etc.) are calculated as the intersection of the sets of security rights available for each component of the system. If a conflict is found involving the obligations and rights of interacting components, it is possible for the system architect to consider an alternative scheme (e.g., using one or more connectors along the paths between the components that act as a security firewall). This means that the architecture and the automated environment together can determine what OA design best meets the problem at hand with available software components. Components with conflicting security licenses do not need to be arbitrarily excluded, but instead may expand the range of possible architectural alternatives if the architect seeks such flexibility and choice.

H. Discussion

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 is a new kind of information structure whose purpose is to declare operational capabilities that express the obligations and rights of users or programs 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 by 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 (see 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 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 Figures 5, 6, and 7, it appears that there may be trade-offs in one layout of security capabilities over another. For example, the layout in Figure 5 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 small-scale 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. 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 design-time, as well as to specify and analyze the security capabilities at build-time 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 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 (Bosch, 2000; Clements & Northrop, 2001). 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 3, which is then instantiated into a

specific software product configuration (as suggested in the build-time architecture shown in Figure 4), then such a reference or design-time architecture as we have presented it here effectively defines an 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 possibly conflict with one another), then we have the situation analogous to what we have presented in this paper. Thus, secure SPL concepts are compatible with secure OA systems that are composed from heterogeneously security licensed components.

I. Conclusion

This paper introduces the concept and initial scheme for systematically specifying and analyzing the security requirements for complex open architecture systems. We argue that such requirements should be expressed as operational capabilities that can be collected and sequenced within a new information structure we call a security license. Such a license expresses security in terms of capabilities that provide users or programs with obligations and rights for how they may access data or control information as well as how they may update or evolve system elements. Thus, these security license rights and obligations play a key role in how and why an OA system evolves in its ecosystem of software component producers, system integrators, and consumers.

We note that changes to the license obligations and rights, whether for control of intellectual property or software security, across versions of components is a characteristic of OA systems whose components are subject to different security requirements or other license restrictions. A structure for modeling software licenses

and automated support for calculating its rights and obligations are needed in order to manage an OA system's evolution in the context of its ecosystem.

We have outlined an approach for achieving these and sketched how they further the goal of reusing components in developing software-intensive systems. Much more work remains to be done, but we believe that this approach turns a vexing problem into one for which workable, as well as robust formal, solutions can be obtained.

Acknowledgments

This research is supported by grant #N00244-10-1-0038 and #N00244-10-1-077 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #0808783 from the U.S. National Science Foundation. No review, approval, nor endorsement is implied.

References

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

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2009a, May). Analyzing software licenses in open architecture software systems. In *Proceedings of the 2nd International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS)*.

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

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2010, May). The challenge of heterogeneously licensed systems in open architecture software ecosystems. In *Proceedings of the 7th Acquisition Research Symposium*. Monterey, CA: Naval Postgraduate School.

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

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

Bosch, J. (2000). *Design and use of software architectures: Adopting and evolving a product-line approach*. Boston, MA: Addison-Wesley.

Breaux, T. D., & Anton, A. I. (2005). Analyzing goal semantics for rights, permissions, and obligations. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering* (pp. 177–188).

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

Clements, P., & Northrop, L. (2001). *Software product lines: Practices and patterns*. Addison-Wesley Professional.

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

Feldt, K. (2007). *Programming Firefox: Building rich internet applications with XU*L. O'Reilly Media.

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

Fontana, R., Kuhn, B. M., Moglen, E., Norwood, M., Ravicher, D. B., Sandler, K., Vasile, J., & Williamson, A. (2008). *A legal issues primer for open source and free software projects*. Software Freedom Law Center.

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

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

Kuhl, F., Weatherly, R., & Dahmann, J. (1999). *Creating computer simulation systems: An introduction to the high level architecture*. Prentice Hall.

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

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

Oreizy, P. (2000). *Open architecture software: A flexible approach to decentralized software evolution* (Doctoral thesis). Irvine, CA: University of California, Irvine.

Rosen, L. (2005). *Open source licensing: Software freedom and intellectual property law*. Prentice Hall.

Scacchi, W., & Alspaugh, T. A. (2008, May). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proceedings of the 5th Annual Acquisition Research Symposium*. Monterey, CA: Naval Postgraduate School.

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

THIS PAGE INTENTIONALLY LEFT BLANK

II. Presenting Software License Conflicts through Argumentation

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

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

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

Abstract

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

A. 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 (Oreizy, 2000). 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. However, 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* (Alspaugh, Asuncion, & Scacchi, 2009) 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 in order to ensure that the desired system rights are available for an acceptable set of obligations.

In our previous work (summarized in Section 4) 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. However, 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, show in what ways the obligations conflict.
- ***Unavailable rights***—For each such right, a traversal identifies the exclusive copyright right that subsumes the right in question, the architectural entity to which the right pertains, and why no right in the entity's license grants the right in question.
- ***Virtual license***—Traversals show the chains of inference by which each right and obligation is entailed by the system architecture, the stated license for each component, and the desired rights for the system as a whole.

The dag calculation algorithm follows the steps of legal 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.

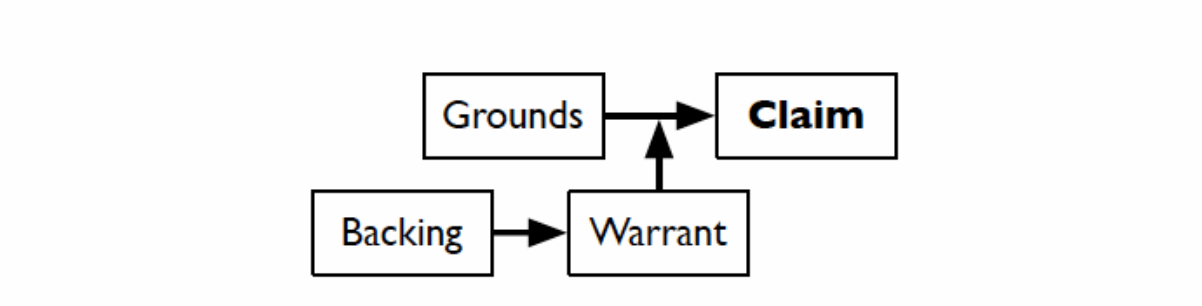

Figure 1. A *Claim*, Supported by *Grounds*, Their Pertinence to the Claim Justified by a *Warrant*, Whose Validity is Supported by *Backing* *(diagram after [14])*

B. Related Work

The most influential approach for structuring legal arguments is that of Toulmin, Rieke, and Janik (1984), 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. This 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; and
4. if the warrant is disputed, then backing supporting it.

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

Hohfeld (1913) sought a theory by which to resolve the imprecise terminology and ambiguous classifications he found in use for legal relationships. In a seminal article published in 1913 and cited to the present day, he set forth a system of eight *jural* relations intended to express and classify all legal relationships between people. The first four regulate ordinary actions and are *right* (may), *no-right* (cannot), *duty* (must), and *privilege* (need not). Each relation has an opposite relation whose sense is its opposite, and a correlative relation whose sense is its complement. We use Hohfeld's first four jural relations as the basis of our representation of the enactable, testable provisions of software licenses (Section 4).

There has been much work on analysis of laws in AI over the past few decades. A widely-cited example is Sergot et al.'s (1986) re-expression of the British Nationality Act as a Prolog program; the resulting program applied the Act to any person's situation and characteristics in order to determine nationality (Sergot et al., 1986).

A number of researchers have used argumentation to guide decision-making, notably Haley, Laney, Moffett, and Nuseibeh (2008) who propose an approach for using satisfaction arguments to evaluate and guide the evolution of security requirements. Decision choices for which no convincing argument is found are set aside in favor of choices for which stronger arguments have been identified.

C. Licensing Background

1. Intellectual Property (IP)

An individual can own a tangible thing and have property rights in it, such as the rights to use it, improve it, sell it or give it away, or prevent others from doing so, subject to some statutory restrictions. Similarly, an individual can own intellectual property (IP) of various types and have specific property rights in the intangible intellectual property, such as the rights to copy, use, change, distribute, or prevent others from doing so, again subject to some statutory restrictions. Software licenses are primarily concerned with copyrights. Copyright is defined by Title 17 of the U.S.

Code and by similar law in many other countries. It grants exclusive rights to the author of an original work in any tangible means of expression; namely the rights to

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

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

2. Software Licenses

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

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

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

Some reciprocal licenses additionally require that software combined with the licensed software (for various definitions of "combined") also be freely licensed

under the same license. We term such licenses propagating; they are also known as *strong copyleft* licenses. Examples are the General Public License versions 2 and 3 (GPLv2, GPLv3; Alspaugh, n.d.).

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).

3. Licenses and Software Architectures

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

Software source code components—These can be

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

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

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

(Rosen, 2005). If proprietary, they often cannot be redistributed, and so such components will be present in the design- and run-time architectures but not in the distribution-time architecture.

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

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

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

Methods of composition—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. Methods of composition affect license obligation propagation, with different methods affecting different licenses. How and to what extent this occurs have not been resolved in court or in practice (Determann, 2006; Stoltz, 2005).

Configured system or subsystem architectures—These are software systems used as atomic components of a larger system. Their internal architecture may contain subcomponents under several licenses, which may affect the rights and obligations for the configured (sub)system and the overall system containing it. To minimize license interaction, a configured system or subsystem architecture may be surrounded by what we term a license firewall (Alspaugh et al., 2009), namely a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of obligations.

4. 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.

D. License Rights and Obligations

In our previous work (Alspaugh et al., 2009) 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 (1913) eight fundamental jural relations, of which we use *right* (may), *duty* (must), *no-right* (must not), and *privilege* (need not; see Figure 2). Each relation has a correlative relation, which in our context relates an obligation to its necessary right:

- if actor A must perform action X, then A requires the correlative right to perform it, expressed as "A may X;"

- if actor A must not perform action X, then A requires the correlative right to not perform it, “A need not X.”

We express rights and obligations as tuples (Figure 3): <actor, modality, action, object, license> The actor is either the “Licensee” or in a few cases “Licensor” for all of the enactable, testable provisions of the licenses we have examined (Alspaugh, Scacchi, & Asuncion, 2010). The modality is may or need not for a right and must or must not for an obligation. The action is a verb phrase acting on an object, describing what may, need not, must, or must not be done. The object is a module of the system or a related artifact such as a source file, the original version, documentation, and so forth. Typically a license right applies to any of a class of objects distributed under the license, such as any binary file or any modified source file; and the right’s obligations will apply to the same object or a related object, such as the right’s object’s sources or the right’s object’s originals. For this reason we term rights and obligations as expressed in a license abstract in contrast to a concrete right or obligation for one specific entity. Some actions are parameterized by a license as well.

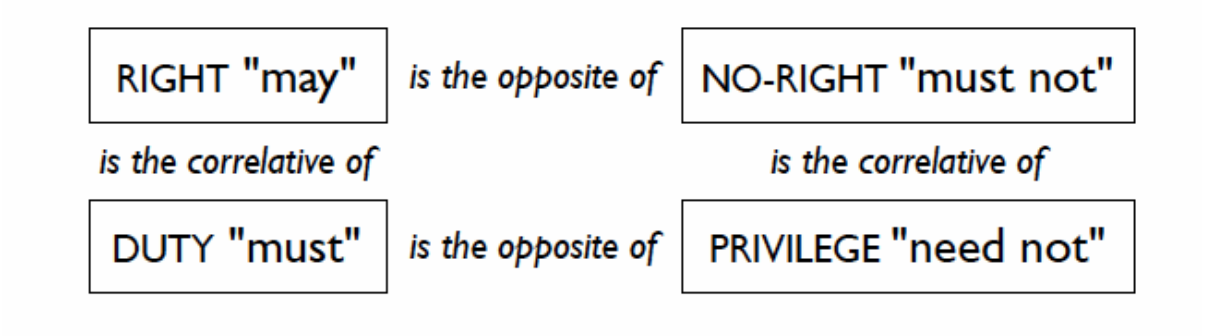

Figure 2. Hohfeld’s Four Basic Relations

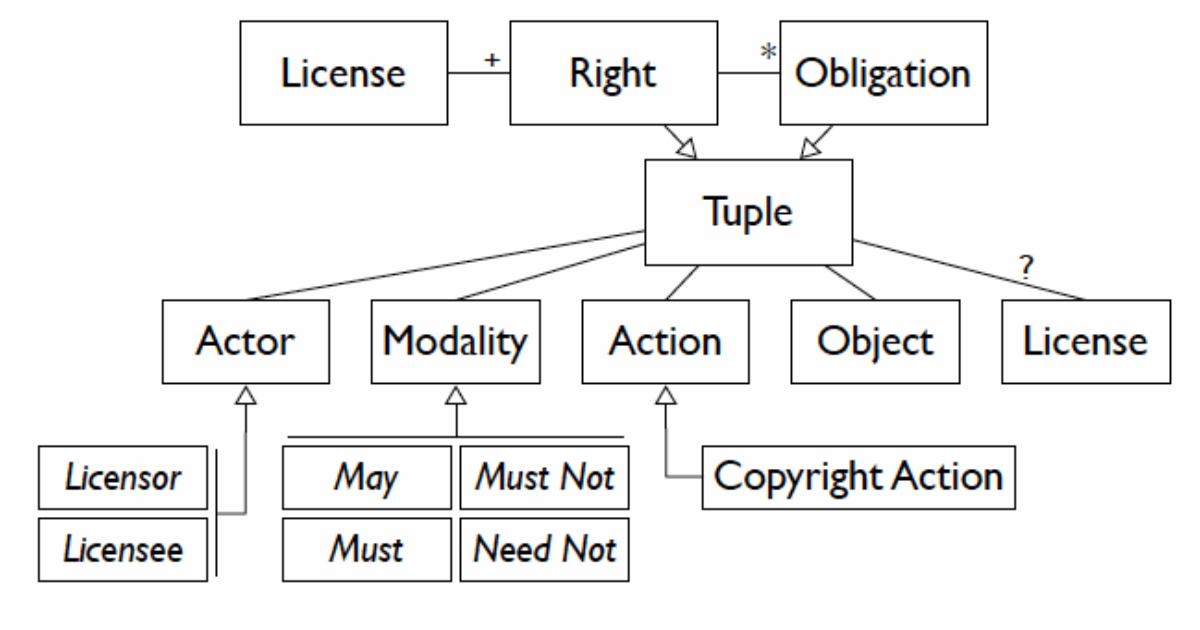

Figure 3. Metamodel for Software Licenses

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 to be granted it, and possibly a set of overall obligations that must be fulfilled for the license as a whole. Figure 4 sketches two rights from GPL version 2.0 (GPLv2), the first with no obligations and the second with three corresponding obligations.

The details of the license specification approach are described in our earlier work (Alspaugh et al., 2009; Alspaugh et al., 2010).

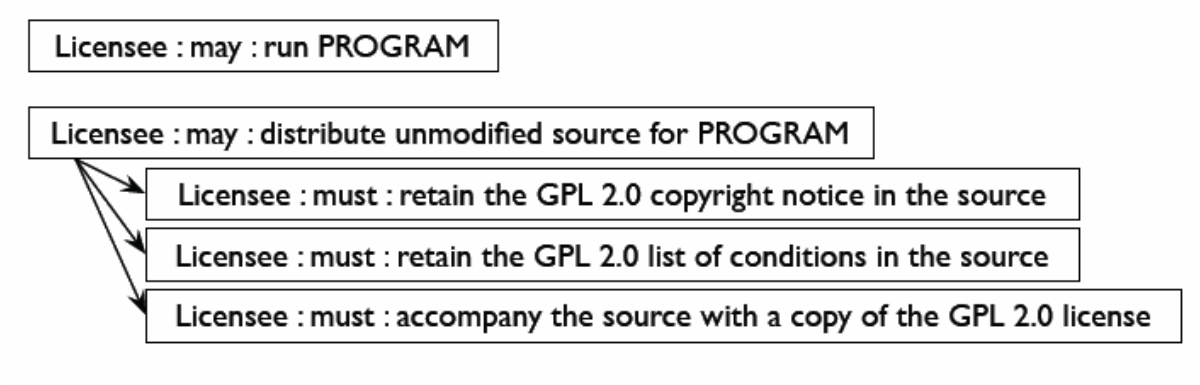

Figure 4. Some Tuples for the GPLv2 License

E. Applying Licenses to Software

1. 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 5 illustrates one step of the application of a license to obtain a desired concrete right r. The license of r's object 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.

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

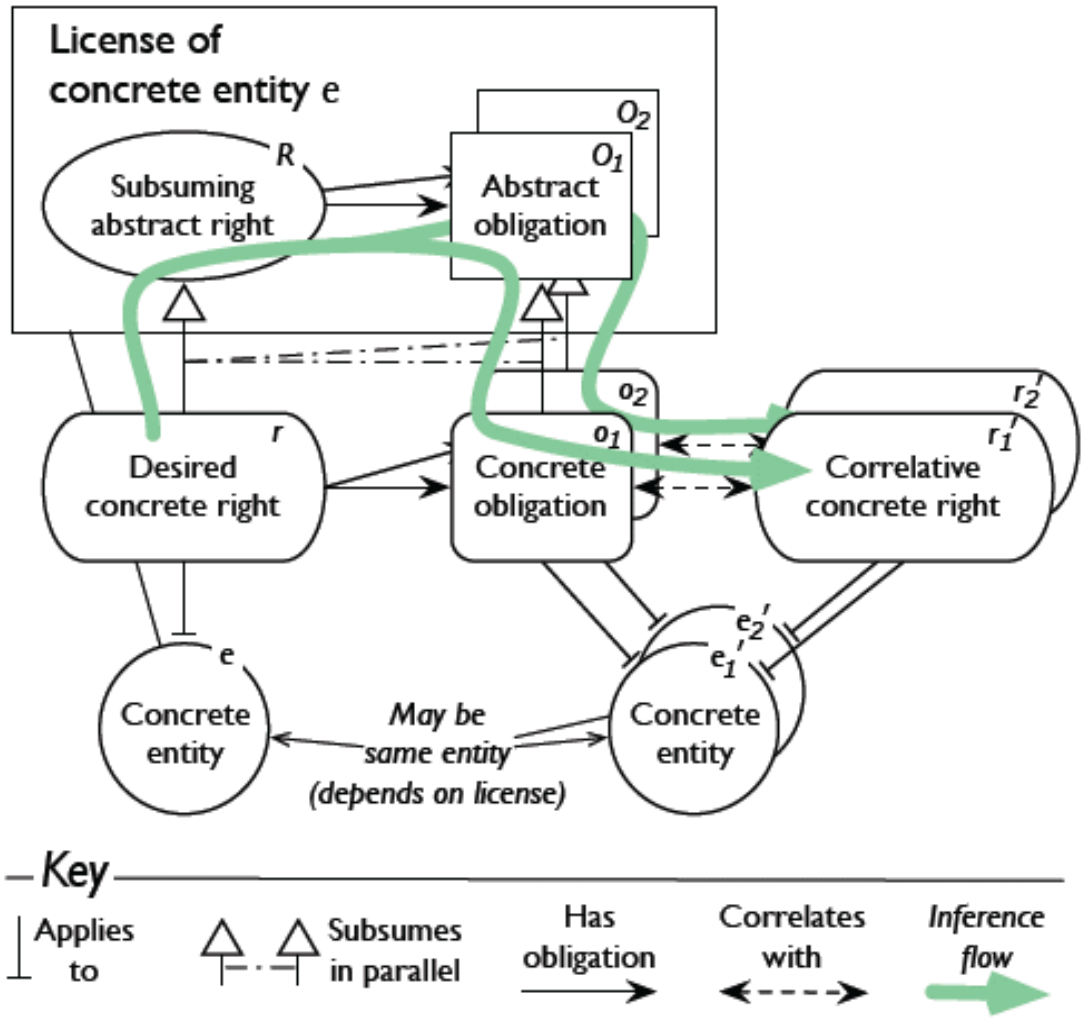

Figure 5. A Step in a Rights/Obligations Inference

Figure 6 shows in Toulmin form a portion of an example inference that produces a conflict, involving a component e1 obtained under GPLv2 and modified, linked to a component e2 obtained under the proprietary Corel Transactional License (CTL; Alspaugh, n.d.). 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 8) these arguments are presented in outline form, with the claim as the root of the outline and its grounds and warrant as its subheads,

to be expanded as desired if further explanation is needed. A typical use would be the following:

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 x2.2{1.bs1).

3. Why can't the WordProcessor component be sublicensed under GPLv2?

4. The WordProcessor component in the architecture has been annotated with the CTL license, and CTL forbids sublicensing under any license (CTL x4{1s1w15).

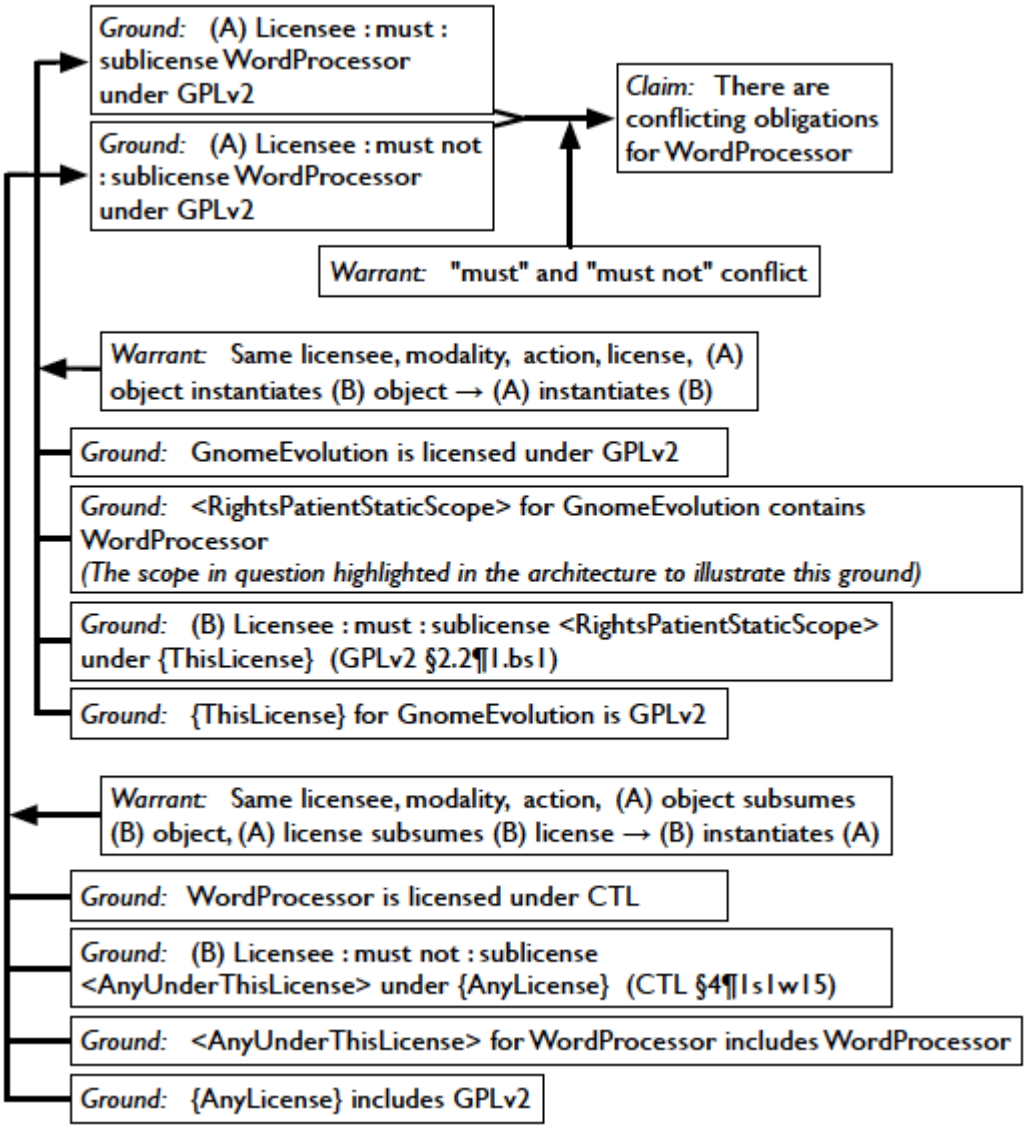

Figure 6. Toulmin-structured Arguments Supporting (and explaining) a typical conflict between obligations for a GPLv2 and a proprietary component

2. Explanation by Argumentation

Figure 7 shows the two explanation flows for a conflict between obligations. Each flow begins at the conflict and explains how one half of the conflicting pair came to be. The connection between the pair is straightforward because 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.

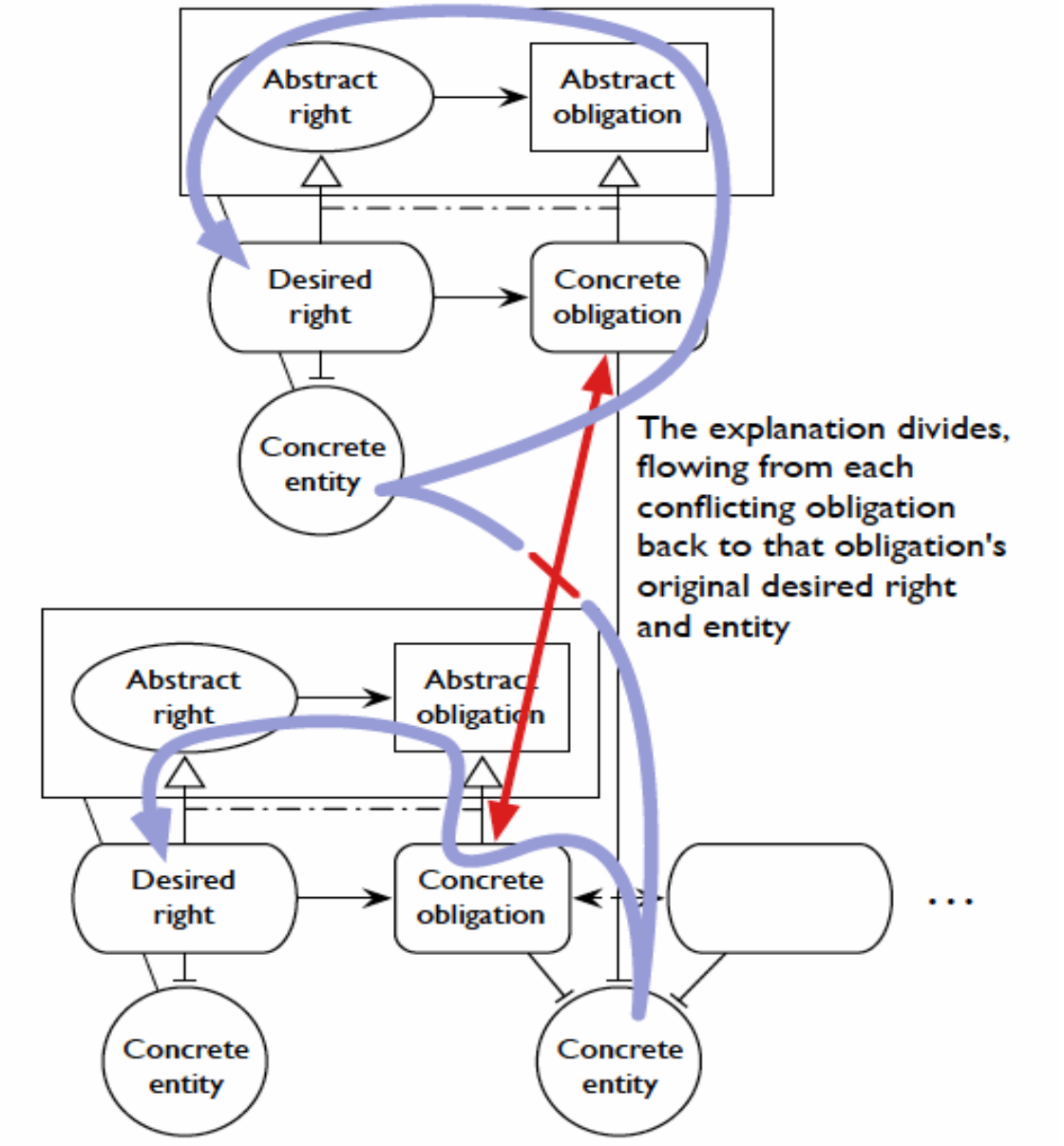

Figure 7. Divided explanation flow for a conflict between two obligations

After examining the kinds of information that are available in the vicinity of a problem (a conflict or unavailable right), we realized that 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 it 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.

3. 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 (Dashofy et al., 2007). The plugin maps features of software architectures onto the license architecture abstraction needed for the virtual license calculation and displays results in the context of the architecture.

The argument grounds drawn from the texts of licenses are implemented through URLs hyperlinking into our collection of software licenses tagged for reference with x-{-sentence word numbers (Alspaugh, n.d.). 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.

F. Conclusion

HtL system design and development provide important benefits but impose new demands difficult to meet using only manual methods and human insight. Our approach for supporting HtL development and acquisition automates the calculation of HtL system virtual licenses. We have integrated it into a software architecture tool so that 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. However, 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 of the available information in a compact, evocative form that is easier to interpret, act on, and verify.

Acknowledgements

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

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

References

Alspaugh, T. A. (n.d.). OSS (and other) licenses, x/{/sentence/word-numbered. Retrieved from http://www.thomasalspaugh.org/pub/osl-sps/

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

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

Dashofy, E., Asuncion, H. U., Hendrickson, S. et al. (2007). Archstudio 4: An architecture-based meta-modeling environment. In *Proceedings of the 28th International Conference on Software Engineering* (Companion volume; pp. 67–68).

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

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

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

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

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

Oreizy, P. (2000). *Open architecture software: A flexible approach to decentralized software evolution* (PhD thesis). Irvine, CA: University of California, Irvine.

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

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

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.

Toulmin, S., Rieke, R., & Janik, A. (1984). *An introduction to reasoning*. New York: Macmillan.

THIS PAGE INTENTIONALLY LEFT BLANK

III. Modding as an Open Source Approach to Extending Computer Game Systems

Walt Scacchi
Institute for Software Research
and
Center for Computer Games and Virtual Worlds
University of California, Irvine
wscacchi@ics.uci.edu

Abstract

This paper examines what is known so far about the role of open source software development within the world of game mods and modding practices. Game modding has become a leading method for developing games by customizing or creating OSS extensions to game software in general, and to proprietary closed source software games in particular. What, why, and how OSS and CSS come together within an application system is the subject for this study. The research method is observational and qualitative so as to highlight current practices and issues that can be associated with software engineering and game studies foundations. Numerous examples of different game mods and modding practices are identified throughout.

A. Introduction

User modified computer games, hereafter *game mods*, are a leading form of user-led innovation in game design and game play experience. But modded games are not standalone systems because they require the user to have an originally acquired or licensed copy of the unmodded game software.

Modding, the practice and process of developing game mods, is an approach to end-user game software engineering (Burnett, Cook, & Rothermel, 2004) that establishes both social and technical knowledge for how to innovate by wresting control over game design from their original developers. At least four types of game mods can be observed: user interface customization, game conversions, machinima,

and hacking closed game systems. Each supports different kinds of open source software (OSS) extensions to the base game or game run-time environment. Game modding tools and support environments that support the creation of such extensions also merit attention. Furthermore, OSS game extensions are commonly applied to either proprietary, closed source software (CSS) games, or to OSS games, but generally more so to CSS games. Why this is so also merits attention. Subsequently, we conceive of game mods as covering customizations, tailorings, remixes, or reconfigurations of game embodiments, whether in the form of game content, software, or hardware denoting our space of interest.

The most direct way to become a game mod developer (a game modder) is through self-tutoring and self-organizing practices. Modding is a form of learning—learning how to mod, learning to be a game developer, learning to become a game content/software developer, learning computer game science outside or inside an academic setting, and more (El-Nasr & Smith, n.d.; Scacchi, 2004). Modding is also a practice for learning how to work with others, especially on large, complex games/mods. Mod team efforts may also self organize around emergent software development project leaders or "want to be" (WTB) leaders, as seen for example in the *Planeshift* (see http://www.planeshift.it/) OSS massively multiplayer online role-playing game (MMORPG) development and modding project (Scacchi, 2004).

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, especially so for games, but also for hardware-centered activities like automobile or personal computer customization. Modders are players of the games they reconfigure, just as OSS developers are also users of the systems that they develop. There is no systematic distinction between developers and users in these communities other than that there are many users/players that may contribute little beyond their usage, word of mouth they share with others, and their demand for more such systems. At

OSS portals like SourceForge.net, the domain of “Games” is the second most popular project category with nearly 42,000 active projects, or 20% of all projects.[1] These projects develop either OSS-based games, game engines, or game development tools/Software Development Kits (SDKs), and all of the top 50 projects have each logged more than one million downloads. Thus, the intersection of games and OSS covers a substantial socio-technical plane, as game modding and traditional OSS development are participatory, user-led modes of system development that rely on continual replenishment of new participants joining and migrating through project efforts, as well as new additions or modifications of content, functionality, and end-user experience (Scacchi, 2002, 2004, 2007). Modding and OSS projects are in many ways experiments to prototype alternative visions of what innovative systems might be in the near future, and so both are widely embraced and practiced primarily as a means for learning about new technologies, new system capabilities, new working relationships with potentially unfamiliar teammates from other cultures, and more (Scacchi, 2007).

Consequently, game modding appears to be (a) emerging as a leading method for developing or customizing game software, (b) primarily reliant on the development and use of OSS extensions as the ways and means for game modding, and (c) overlapping a large community of OSS projects that develop computer game software and tools that has had comparatively little study. As such, the research questions that follow are why do these conditions exist, how have they emerged, and how are they put into practice in different game modding efforts.

This paper seeks to examine what is known so far about game mods and modding practices. The research method in this study is observational and qualitative. It seeks to snapshot and highlight current practices that can be

1 See http://www.sourceforge.net/softwaremap/index.php, accessed 15 April 2011. The number one category of projects is for “Development” with more than 65K OSS projects, out of 210K projects. So OSS Development and OSS Games together represent half of the projects currently hosted on SourceForge.

associated with software engineering and game studies, as well as how these practices may be applied in CSS versus OSS game modding. Numerous examples of different game mods and modding practices are identified throughout in order to help establish an empirically grounded baseline of observations from which further studies can build or refute. Furthermore, the four types of game mods and modding practices identified in this paper have been employed first-hand in game development projects led or produced by the author. Such observation can subsequently serve as a basis for further empirical study and technology development that ties together computer games, OSSD, software engineering, and game studies (Scacchi, 2002, 2004, 2007, 2010).

B. Related Work

Two domains of research inform the study here: software extension within the field of software engineering and modding as a cultural practice within game studies. Each is addressed in turn.

1. Software Extension

Game mods embody different techniques and mechanisms for software extension. However, the description of game mods and modding is often absent of its logical roots or connections back to software engineering. As suggested, mods are extensions to existing game software systems, so it is appropriate to review what we already know about software extensions and extensibility.

Parnas (1979) provides an early notion of software extension as an expression of modular software design. Accordingly, modular systems are those whose components can be added, removed, or updated while satisfying the original system functional requirements. Such concepts in turn were integrated into software architectural design language descriptions and configuration management tools (Narayanaswamy & Scacchi, 1987). However, reliance on explicit software architecture descriptions is not readily found in either conventional game or mod development. Hentonnen, Matinlassi, Niemela, and Kanstren (2007) examine how

software plug-ins support architectural extension, while Leveque, Estublier, and Vega (2009) investigate how extension mechanisms like views and model-based systems support extension, also at the architectural level. Last, the modern Web architecture is itself designed according to principles of extensibility through open APIs, migration across software versions, network data content/hypertext transfer protocols, and representational state transfer (Fielding & Taylor, 2002). Mod-friendly networked multi-player games often take advantage of these capabilities.

Elsewhere, Batory, Johnson, MacDonald, and von Heeder (2002) describe how domain-specific languages (for scripting) and software product lines support software extension, and how now such techniques are used in games that are open for modding. Next, OSS development, as a complementary approach to software engineering, relies on OSS code and associated online artifacts that are open for extension through modification and redistribution of their source representations (Scacchi, 2007). Finally, other techniques to extend the functionality or operation of an existing CSS system may include unauthorized modifications that might go beyond what the end-user license agreement might allow, and so appear to fall outside of what software engineering might anticipate or encourage. These include extensions via hacking methods like code injection or hooking, whose purpose is to gain/redirect control of normal program flow through overloading or intercepting system function calls or provide a hidden layer of interpretation, which allow for “man in the middle” interventions. Thus, software extensions and extensibility is a foundational concept in software engineering, as well as foundational to the development of game mods. However, the logical connections and common/uncommon legacy of game modding, OSS development, and software engineering remain under specified, which this paper begins to address.

2. Modding as Cultural Practice

Game modding is a practice for user content creation that creates/networks not only game mods but game modders. Within anthropological, behavioral, and sociological studies of computer game play, modding has been studied as an

emerging cultural practice that mediates both game play and player interaction with other players (including the game's developers). In some early studies, modding has been designated as a form of “playbour” whereby player actions to create game extensions for use by other players is observed as a form of unpaid (or underpaid) labor that primarily benefits the financial and property interests of game development corporations or hegemonic publishers (Kücklich, 2005; Postigo, 2007; Yee, 2006).

Game modding also modifies or transforms game play experience, since what is play and what is experience(d) are culturally situated. Examples of this may include single player games being modded into multi-player games. Therefore, the experience of single player versus the game environment is transformed into other situations including player versus player, multi-player group play, or team versus team play. Similarly, the modding of games to enable experiences other than expected game play, like using a modded game for storytelling or film-making experiences is also a practice of growing interest, with the emergence of a distinguishable community of gamer-filmmakers who produce *machinima* (described in Section 3) as either a literary medium or an art form (Kelland, 2011; Lowood & Nitsche, 2011; Marino, 2004).

Other studies have observed that user/modders also benefit from modding as a way to achieve a sense of creative ownership and meaning in the modded games that they share and play with others (Postigo, 2008; Scacchi, 2002, 2004; Sotamaa, 2010), and that game mods and modding practices become central elements in what constitutes play with and through games (Taylor, 2009). Finally, as already observed, OSS project portals like SourceForge host thousands of OSS game development projects that develop and deploy role-playing games (4,300 projects), simulation-based games (2,600), board games (2,300), side-scrolling/arcade games (2,000), turn-taking strategy games (1,700), multi-user dungeons or text-based adventure/virtual worlds (1,600), first-person shooters (1,600), MMORPG (600) and more. Thus, development of OSS games and related game development tools can be recognized as a central element in the cultural world of computer games and

game development as well as in the world of OSS development (Scacchi, 2002, 2004, 2007).

C. Four Types of Game Mods

At least four types of game mods are realized through OSS development practices. These include (a) user interface customizations and agents, (b) game conversions, (c) machinima, and (d) hacking closed source game systems. Each type of game mod is examined in turn and each is facilitated (or prohibited) according to its copyright license.

1. User Interface Customizations and Agents

User interfaces to games embody the practice and experience of interfacing users (game players) to both the game system and the play experience designed by the game's developers. Game developers act to constrain and govern what users can do and what kinds of experiences they can realize. Some users in turn seek to achieve a form of competitive advantage during game play by modding the user interface software for their game when so enabled by game developers. These mods acquire or reveal additional information that users believe will help their play performance and experience. User interface add-ons subsequently act as the medium through which game development studios support game product customization, which is a strategy for increasing end-user satisfaction and thus the likelihood of product success (Burnett et al., 2004).

Three kinds of user interface customizations can be observed. First and most common, is the player's ability to select, attire, or accessorize a *player's in-game identity*. Second, is for players to customize *the color palette and representational framing borders* of the their game display within the human-computer interface, much like what can also be done with Web browsers (e.g, Firefox 4 “personas” and “themes”) and other end-user software applications. Third, are *user interface add-on modules* that modify the player's in-game information management dashboard, but do not modify the underlying game play rules or functions. These add-ons provide

additional information about game play state that may enhance the game play experience as well as increase a player's sense of immersion or omniscience within the game world through perceptual expansion. This in turn enables awareness of game events not visible in the player's pre-existing in-game view. Furthermore, some add-on facilities (e.g., those available with the proprietary *World of Warcraft* MMORPG, scripted in the LUA language) accommodate the creation of automated agent scripts that can read/parse data streamed to the UI within an existing or other add-on dashboard component and then provide some additional value-added play experience, such as sending out messages or status reports to other players automatically. Such add-on agents modify or reconfigure the end-user play experience rather than the core functionality or play mechanics available to all of the game's other players. Consequently, the first two kinds of customizations result from meta-data selections within parametric system functions, whereas the third represents a traditional kind of user-created modular extension; one that does not affect the pre-existing game's functional requirements, nor one included in the operational source code base during subsequent system builds or releases, unless the extension does alter the software's requirements (e.g., by introducing a new security vulnerability or exploit that must be subsequently prevented).

2. Game Conversions

Game conversion mods are perhaps the most common form of game mods. Most such conversions are partial, in that they add or modify (a) in-game characters including user-controlled character appearance or capabilities, opponent bots, cheat bots, and non-player characters, (b) play objects like weapons, potions, spells, and other resources, (c) play levels, zones, maps, terrains, or landscapes, (d) game rules, or (e) play mechanics. Some more ambitious modders go as far as to accomplish (f) total conversions that create entirely new games from existing games of a kind not easily determined from the original game. For example, one of the most widely distributed and played total game conversions is the *Counter-Strike* (CS) mod of the *Half-Life* (HL) first-person action game from Valve Software. As the success of

the CS mod gave rise to millions of players preferring to play the mod over the original HL game, other modders began to access the CS mod to further convert in part or full, 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 that is available to players who acquire a licensed copy of the HL product family. Valve has since marketed a number of CS variants that have sold over 10 million copies as of 2008; thus, denoting the most successful game conversion mod, as well as the most lucrative in terms of subsequent retail sales derived from a game mod.

Another example is found in games converted to serve a purpose other than entertainment, such as the development and use of games for science, technology, and engineering applications. For instance, the *FabLab* game (Scacchi, 2010) is a conversion of the *Unreal Tournament 2007* retail game from a first-person shooter to a simulator for training semiconductor manufacturing technicians in diagnosing and treating potentially hazardous materials spills in a cleanroom environment. This conversion was not readily anticipated by knowledge of the Unreal games or underlying game engine, although it maintains operational compatibility with the Unreal game itself. Therefore, game conversions can re-purpose the look, feel, and intent of a game across application domains, while maintaining a common software product line (Batory et al., 2002).

Finally, it is common practice that the underlying game engine has one set of license terms and conditions to protect original work (e.g., no redistribution), while game mods can have a different set of terms and conditions as a derived work (e.g., redistribution allowed only for a game mod, but not for sale). In this regard, software licenses embody the business model that the game development studio or publisher seeks to embrace, rather than just a set of property rights and constraints. For example, in *Aion*, an MMORPG from the South Korean game studio NCSoft, no user created mods or user interface add-ons are allowed. Attempting to incorporate such changes would conflict with its end-user license agreement (EULA) and

subsequently put such user-modders at risk of losing their access to networked *Aion* multi-player game play. In contrast, the MMORPG *World of Warcraft* (WoW) allows for UI customization mods and add-ons only, but no other game conversions, no reverse engineering of the game engine, and no activity intended to bypass WoW's encryption mechanisms. Additionally, in one more variation, for games like *Unreal Tournament*, *Half-Life*, *NeverWinterNights*, *Civilization*, and many others, the EULAs encourage modding and the free redistribution of mods without fee to others who must have a licensed copy of the proprietary CSS game, but do not allow reverse engineering or redistribution of the CSS game engine required to run the OSS mods. This restriction in turn helps game companies realize the benefit of increased game sales by players who want to play with known mods rather than with the un-modded game as sold at retail. Thus, mods help improve games' software sales, revenue, and profits for the game development studio, publisher, and retailer as well as enable new modes of game play, learning, and skill development for game modders.

3. Machinima

Machinima can be viewed as the product of modding efforts that intend to modify the visual replay of game usage sessions. Machinima employ computer games as their creative media, such that these new media are mobilized for some other purpose (e.g., creating online cinema or interactive art exhibitions). Machinima focuses attention to playing and replaying a game for the purpose of storytelling, movie making, or retelling of daunting or high efficiency game play/usage experiences (Lowood & Nitsche, 2011; Marino, 2004). Machinima is a form of modding the experience of playing a specific game by recording its visual play session history so as to achieve some other ends beyond the enjoyment (or frustration) of game play. These play-session histories can then be further modded via video editing or remixing with other media (e.g., adding music) to better enable cinematic storytelling or creative performance documentation. Machinima is a kind of play/usage history process re-enactment (cf. Scacchi, 1998), whose purpose may be documentary (replaying what the player saw or experienced during a play session)

or cinematic (creatively steering a play session so as to manifest observable play process enactments that can be edited and remixed off-line to visually tell a story). Thus, machinima mods are a kind of extension of the game software use experience that is not bound to the architecture of the underlying game software system, except for how the game facilitates a user's ability to structure and manipulate emergent game play to realize a desired play process enactment history.

4. Hacking Closed Game Systems

Hacking a closed game system is a practice whose purpose oftentimes seems to be in direct challenge to the authority of commercial game developers that represent large, global corporate interests. Hacking proprietary game software is often focused not so much on how to improve competitive advantage in multi-player game play, but instead is focused on expanding the range of experiences that users may encounter through use of alternative technologies (Huang, 2003; Scacchi, 2004). For example, Huang's (2003) study instructs readers in the practice of “reverse engineering” as a hacking strategy to understand both how a game platform was designed and how it operates in fine detail. This in turn enables reconfiguration of new innovative modifications or original platform designs, such as installing and running a Linux operating system (instead of Microsoft's proprietary CSS offering). Although many game developers seek to protect their intellectual property (IP) from reverse engineering through end-user license agreements (EULAs), whose terms attempt to prohibit such action under threat of legal action, reverse engineering is not legally prohibited. Consequently, the practice of modding closed game consoles/systems is often less focused on enabling players to achieve competitive advantage when playing retail computer games, but instead may encourage those few so inclined for how to understand and ultimately create computing innovations through reverse engineering or other modifications.

Closed game system modding is a style of software extension used by game modders who are willing to forego the “protections” and quality assurances that closed game system developers provide, in order to experience the liberty, skill,

knowledge acquisition, conceptual appropriation ("owned"), and potential to innovate that mastery of reverse engineering affords. Consequently, players/modders who are willing to take responsibility for their actions (and not seek to defraud game producers due to false product warranty claims or copyright infringement), can enjoy the freedom to learn how their gaming systems work in intimate detail and potentially learn about game system innovation through discovery and reinvention with the support of others who are like-minded (cf. Scacchi, 2004). Proprietary game development studios may sometimes allow for such mod-based infringement of their games. For example, the team of modders behind the hacking and conversion of the single-player CSS game, *Grand Theft Auto*, have produced an OSS (now GPL'd) game mod using code injection and hooking cheating methods to realize a networked multi-player variant called *Multi Theft Auto,* that Rockstar Games has chosen not to prosecute for potential EULA violation, but instead to embrace as GTA fan culture (Wen, 2011). Nonetheless, large corporate interests may assert that their IP rights allow them to install CSS root kits that collect potentially private information, or that prevent the reactivation of previously available OSS (e.g., the Linux Kernel on the Sony PS3 game console[2]) that game system hackers seek to undo.

Finally, games are one of the most commonly modified types of proprietary CSS that are transformed into "pirated games" that are "illegally downloaded." Such game modding practices are focused on engaging a kind of meta-game that involves hacking into and modding game IP from closed to (more) open. Thus game piracy has become recognized as a collective, decentralized, and placeless endeavor (i.e., not a physical organization) that relies on torrent servers as its underground distribution venue for pirated game software. As recent surveys of torrent-based downloads reveal, in 2008 the top 10 pirated games represented about 9 million downloads, whereas in 2009 the top 5 pirated games represented more than 13 million downloads, and in 2010 the top 5 pirated games approached 20 million, all

2 For details, see http://en.wikipedia.org/wiki/George_Hotz#Hacking_the_PlayStation_3

suggesting a substantial growth in interest in and access to such modded game products.[3] Thus, we should not be surprised by the recent efforts of game system hackers that continue to demonstrate the vulnerabilities of different hardware and software-based techniques to encrypt and secure closed game systems from would be hackers. However, it is also very instructive to learn from these exploits how difficult it is to engineer truly secure software systems, whether or not such systems are games or some other type of application or package.

D. Game Modding Software Tools and Support

Games are most often modded with tools providing access to unencrypted representations of game software or game platforms. Such a representation is accessed and extended via a domain-specific (scripting) language. Although it might seem the case that game vendors would seek to discourage users from acquiring such tools, a widespread contrary pattern is observed.

Game system developers are increasingly offering software tools for modifying the games they create or distribute as a way to increase game sales and market share. Game/domain-specific Software Development Kits (SDKs) provided to users by game development studios represent a contemporary business strategy for engaging users to help lead product innovation from outside the studio. Once Id Software, maker of the *DOOM* and *Quake* game software product line, and Epic Games, maker of the *Unreal* software game product line, started to provide prospective game players/modders with software tools that would allow them to edit game content, play mechanics, rules, or other functionality, other competing game development studios were pressured to make similar offerings or face a possible competitive disadvantage in the marketplace. However, the CSS versions of these

3 For 2008, see http://torrentfreak.com/top-10-most-pirated-games-of-2008-081204/
For 2009, see http://torrentfreak.com/the-most-pirated-games-of-2009-091227/
For 2010, see http://torrentfreak.com/call-of-duty-black-ops-most-pirated-game-of-2010-101228/

tools do not provide access to the underlying source code that embodies the proprietary game engine—a large software program infrastructure that coordinates computer graphics, user interface controls, networking, game audio, access to middleware libraries for game physics, and so forth. However, the complexity and capabilities of such a tool suite mean that any one person, or better said, any game development or modding team, can now access modding tools or SDKs to build commercial quality CSS games through OSS extensions. However, mastering these tools appears to be an undertaking likely to be of interest only to highly committed game developers who are self-supported or self-organized.

In contrast to game modding platforms provided by game development studios, there are also alternatives provided by the end-user community. One approach can be seen with facilities provided in meta-mods like *Garry's Mod* or the *AMX Mod X* mod-making package. Modders can use these packages to construct a variety of plug-ins that provide for the development of in-game contraptions as game UI agents or user created art works, or to otherwise create comic books, program game conversions, and other kinds of user created content. However, both packages require that you own a licensed CSS game like *Counter-Strike: Source*, *Half-Life2* or *Day of Defeat: Source* from Valve Software.

A different approach to end-user game development platforms can be found arising from OSS games and game engines. The *DOOM* and *Quake* games and game engines were released as free software subject to the GPL once they were seen by Id Software as having reached the end of their retail product cycle. Thousands of games/engines, as already observed, have been developed and released for download. Some started from the OSS that was previously the CSS platform of the original games. However, the content assets (e.g., in-game artwork) for many of these CSS- then-OSS games are not covered by the GPL, and so user-developers must still acquire a licensed copy of the original CSS game if its content

is to be reused in some way.[4] Nonetheless, some variants of the user-created GPL'd games now feature their own content that is limited/protected by Creative Commons licenses.

E. Opportunities and Constraints for Modding

Game modding demonstrates the practical value of software extension as a user-friendly approach to customizing software. Such software can extend games open to modding into diverse product lines that flourish through reliance on domain-specific game scripting languages and integrated SDKs. Modding also demonstrates the success of end-users learning how to extend software to create custom user interface add-ons, system conversions, and replayable system usage videos as well as to discover security vulnerabilities. Therefore game modding represents a viable form of end-user engineering of complex software that may be transferable to other domains.

Modding is a form of OSS-enabled collaboration. It is collaboration at a distance, where the collaborators, including the game developers and game users, are distant in space and time from each other yet they can interact in an open but implicitly coordinated manner through software extensions. Comparatively little explicit coordination arises, except when CSS game developers seek to embrace and encourage the creation of OSS game mods that rely on the proprietary CSS game engine (and also SDK) as a way to grow market share and mid share for the proprietary engine as a viable strategy for entry into the game industry.

However, mods are vulnerable to evolutionary system version updates that can break the functionality or interface on which the mod depends. This can be viewed as the result of inadequate software system design practice, such that existing system modularization did not adequately account for software extensions that end-users seek, or else the original developer wanted to explicitly prohibit end-

4 For example, see http://assault.cubers.net/docs/license.html , accessed 13 April 2011.

users from making modifications that transform game play mechanics/rules or unintentionally allow for modification or misappropriation of copy-protected code or media assets.

Last, one of the key constraints on game modding in particular, and software extension in general, are the rights and obligations that are expressed in the original software EULA. Mods tend to be licensed using OSS or freeware licenses that allow for access, study, modification, and redistribution rather than using free software licenses (e.g., GPLv2 or GPLv3). Software extensions that might be subject to a reciprocal GPL-style license require that the base/original software system incorporate an explicit software architectural design that requires the propagation of reciprocal rights across an open interface, except through an LGPL software shim (Alspaugh, Asuncion, & Scacchi, 2009). Otherwise, the scope of effectiveness and copyright protections of either free or non-free software (or related media assets) cannot be readily determined, and thus may be subject to copyright infringement or licenses non-compliance allegations. They may also be treated as social transgressions within a community of modders whose perceived ownership of the game mods demands the respect and honor of a virtual license that may or may not be legally valid (Alspaugh, Scacchi, & Asuncion, 2010). As the OSS community has long recognized, software rights and freedoms are expressed through IP licenses that ensure whether or not a person has the right to access, study, modify, and redistribute the modified software as long as the obligation to include a free software license is included that restates these rights in unalterable form, is included with the OSS code and its modified distributions.

F. Conclusions

Modding is emerging as a viable approach for mixing proprietary CSS systems with OSS extensions. The result is modded systems that provide the benefits of OSSD to developers of proprietary CSS systems and to end-users who want additional functionality of their own creation or from others they trust and seek to interact with through game play.

In contrast, modding is not so good for protecting software and media/content copyrights. Modding tests the limits of software/IP copyright practices. Some modders want to self-determine what copy/modding rights they have or not, and sometimes they act in ways that treat non-free software and related media as if it were free software. Who owns what, and which copy rights or obligations apply to that which is modded, are core socio-technical issues when engaging in modding.

This study helps to demonstrate that game modding is becoming a leading method for developing or customizing game software, whether based on proprietary CSS or OSS game systems. OSS-based software extensions are the leading ways and means for modding game-based user interfaces, for converting games from one style/genre to another, for recording game play sessions for cinematic production and replay, and for hacking closed source game systems. Finally, the development of computer game software and tools itself represents a large community of OSS projects that has had comparatively little study, and thus merits further attention as its own cultural world as well as one for OSS development. This last consideration may be important because other empirical studies of OSS development that rely on data from SourceForge will increasingly include OSS game projects within large project samples. Therefore, this study has begun to address why and how these conditions have emerged and how they are put into practice in different game modding efforts. Future study should also consider whether and how modding might be applied and adopted in other application domains where CSS can be extended through OSS mods.

Acknowledgments

The research described in this paper has been supported by grants #0808783 and #1041918 from the National Science Foundation, and grant #N00244-10-1-0077 from the Naval Postgraduate School. No review, approval, or endorsement is implied. The anonymous reviewers also provided helpful suggestions for improving this paper.

References

Alspaugh, T. A., Asuncion, H. A., & Scacchi, W. (2009, September). Intellectual property rights requirements for heterogeneously licensed systems. In *Proceedings of the 17th International Conference on Requirements Engineering* (*RE09*; pp. 24–23). Atlanta, GA. Retrieved from http://www.ics.uci.edu/~wscacchi/Papers/New/Alspaugh-Asuncion-Scacchi-RE09.pdf

Alspaugh, T. A., Scacchi, W., & Asuncion, H. A. (2010, November). Software licenses in context: The challenge of heterogeneously licensed systems. *Journal of the Association for Information Systems*, *11*(11), 730–755. Retrieved from http://www.ics.uci.edu/~wscacchi/Papers/New/Scacchi-Alspaugh-Asuncion-JAIS.pdf

Batory, D., Johnson, C., MacDonald, B., & von Heeder, D. (2002). Achieving extensibility through product lines and domain specific languages: A case study. *ACM Trans. Software Engineering and Methodology*, *11*(2), 191–214.

Burnett, M., Cook, C., & Rothermel, G. (2004). End-user software engineering. *Communications ACM*, *47*(9), 53–58.

El-Nasr, M. S., & Smith, B. K. (n.d.). Learning through game modding. *ACM Computers in Entertainment*, *4*(1), Article 3B.

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

Huang, A. (2003). *Hacking the Xbox: An introduction to reverse engineering.* San Francisco, CA: No Starch Press.

Henttonen, K., Matinlassi, M., Niemela, E., & Kanstren, T. (2007). Integrability and extensibility evaluation in software architectural models—A case study. *The Open Software Engineering Journal, 1*(1), 1–20.

Kelland, M. (2011). From game mod to low-budget film: The evolution of machinima. In H. Lowood & M. Nitsche (Eds.), *The Machinima Reader* (pp. 23–36). Cambridge, MA: MIT Press.

Kücklich, J. (2005). Precarious playbour: Modders and the digital games industry. *Fiberculture*, *5*. Retrieved from http://journal.fibreculture.org/issue5/kucklich.html

Leveque, T., Estublier, J., & Vega, G. (2009). Extensibility and modularity for model-driven engineering environments. In *Proceedings of the 16th IEEE Conference on Engineering Computer-Based Systems* (*ECBS 2009*; pp. 305–314).

Lowood, H., & Nitsche, M. (Eds.). (2011). *The machinima reader*. Cambridge, MA: MIT Press.

Marino, P. (2004). *3D game-based filmmaking: The art of machinima*. Scottsdale, AZ: Paraglyph Press.

Narayanaswamy, K., & Scacchi, W. (1987). Maintaining evolving configurations of large software systems. *IEEE Trans. Software Engineering, SE-13*(3), 324–334.

Parnas, D.L. (1979). Designing software for ease of extension and contraction. *IEEE Trans. Software Engineering, SE-5*(2), 128–138.

Postigo, H. (2007). Of mods and modders: Chasing down the value of fan–based digital game modifications. *Games and Culture*, *2*(4), 300–313.

Postigo, H. (2008). Video game appropriation through modifications: Attitudes concerning intellectual property among modders and fans. *Convergence*, *14*(1), 59–74.

Scacchi, W. (1998). Modeling, integrating, and enacting complex organizational processes. In K. Carley, L. Gasser, & M. Prietula (Eds.), *Simulating organizations: Computational models of institutions and groups* (pp. 153–168). Cambridge, MA: MIT Press.

Scacchi, W. (2002, February). Understanding the requirements for developing open source software. *IEE Proceedings—Software Engineering, 149*(1), 24–39. Revised version in K, Lyytinen, P. Loucopoulos, J. Mylopoulos, and W. Robinson (Eds.), *Design Requirements Engineering: A Ten-Year Perspective*, LNBIP 14, Springer-Verlag, 467-494, 2009.

Scacchi, W. (2004, January/February). Free/open source software development practices in the game community. *IEEE Software*, *21*(1), 59–67.

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

Scacchi, W. (2010). Game-based virtual worlds as decentralized virtual activity systems. In W. S. Bainbridge (Ed.), *Online worlds: Convergence of the real and the virtual* (pp. 225–236). New York: Springer.

Sotamaa, O. (2010). When the game is not enough: Motivations and practices among computer game modding culture. *Games and Culture*, *5*(3), 239–255.

Taylor, T. L., (2009). The assemblage of play. *Games and Culture*, *4*(4), 331–339.

Wen, H. (2005, May). Multi theft auto: Hacking multi-player into Grand Theft Auto with open source. *OSDir*. Retrieved from http://osdir.com/Article4775.phtml. Also see http://www.mtavc.com/ and http://en.wikipedia.org/wiki/Multi_Theft_Auto.

Yee, N. (2006). The labor of fun: How video games blur the boundaries of work and play. *Games and Culture*, *1*(1), 68–71.

IV. Modding as a Basis for Developing Game Systems

Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
wscacchi@ics.uci.edu

Abstract

This paper seeks to briefly examine what is known so far about game mods and modding practices. Game modding has become a leading method for developing games by customizing extensions to game software. The research method in this study is observational and qualitative, so as to highlight current practices and issues that can be associated with software engineering foundations. Numerous examples of different game mods and modding practices are identified throughout.

Categories and Subject Descriptors

D.2 [Software Engineering]: software development, software architecture, design methodology

General Terms

Design, Human Factors.

Keywords: Computer games, software extension, game modding

GAS'11, May, 2011, Waikiki, Honolulu, HI, USA.

A. Introduction

User modified computer games, hereafter *game mods*, are a leading form of user-led innovation in game design and game play experience. But modded games are not new, clean-sheet standalone systems, as they require the user to have an originally acquired or authorized copy of the unmodded game.

Modding, the practice and process of developing game mods, is typically a "Do It Yourself" (DIY) approach to end-user game software engineering (Burnett, Cook, & Rothermel, 2004) that can establish both social and technical knowledge for how to innovate by wresting control over game design from their original developers. At least four types of game mods can be observed: user interface customization, game conversions, machinima, and hacking closed game systems. Each enables different kinds of extension to the base game or game run-time environment. Game modding tools and support environments that support the creation of such extensions also merit attention. Subsequently, we conceive of game mods as covering customizations, tailorings, and remixes—that is, *software extensions*—of game embodiments, whether in the form of game content, software, or hardware denoting our space of interest.

The most direct way to become a game modder is through self-tutoring and self-organizing practices. Modding is a form of learning—learning how to mod, learning to be a game developer, learning to become a game content/software developer, learning computer game science outside or inside an academic setting, and more (El-Nasr & Smith, n.d.). Modding is also a practice for learning how to work with others, especially on large, complex games/mods. Mod team efforts may also self organize around emergent software development project leaders or "want to be" (WTB) leaders, as seen for example in the *Planeshift* open source MMOG development/modding project (Scacchi, 2004).

Game mods, modding practices, and modders are in many ways quite similar to their counterparts in the world of free/open source software development

(FOSSD). Modding is to games, like FOSSD is to software—they are increasingly becoming a part of mainstream technology development culture and practice. Modders are players of the games they construct, just like FOSS developers are also users of the systems they develop. There is no systematic distinction between developers and users in these communities, other than that there are users/players that may contribute little beyond their usage, word of mouth they share with others, and their demand for more such systems. FOSSD portals like SourceForge.com, as of January 2011, indicate that the domain of "games" appears as the third most popular project category with over 23,000 active projects. These projects develop either FOSS-based games, game engines, or game tools/Software Development Kits (SDKs), and all of the top 50 projects each have logged more than 1 million downloads. Thus, the intersection of games and FOSS covers a substantial social and technological plane, as both modding and FOSS development are participatory, user-led modes of system development that rely on the continual replenishment of new participants joining and migrating through project efforts as well as new additions or modifications of content, functionality, and end-user experience (Scacchi, 2002, 2004, 2007). Modding and FOSSD projects are in many ways experiments to prototype alternative visions of what innovative systems might be in the near future, and so both are widely embraced and practiced primarily as a means for learning about new technologies, new system capabilities, new working relationships with potentially unfamiliar teammates from other cultures, and more (cf. Scacchi, 2007).

Consequently, game modding can be recognized as a leading method for developing or customizing game software. And software extensions are the leading ways and means for game modding.

This paper seeks to briefly examine what is known so far about game mods and modding practices. The research method in this study is observational and qualitative so as to highlight current practices and issues that can be associated with software engineering foundations. Numerous examples of different game mods and

modding practices are identified throughout to help distinguish empirically grounded observation from conjecture. All of the types of game mods and modding practices identified in this paper have been employed first-hand by game development projects led or produced by the author. Such observation has subsequently served as a basis for further empirical study and technology development that ties together computer games and software engineering (Scacchi, 2002, 2004, 2007).

B. Software Extension

Game mods embody different techniques and mechanisms for software extension. However, the description of game mods and modding is often absent of its logical roots or connections back to software engineering. As suggested, mods are extensions to existing game software systems, so it is appropriate to review what we already know about software extensions and extensibility.

Parnas (1979) provides an early notion of software extension as an expression of modular software design. Accordingly, modular systems are those whose components can be added, removed, or updated while satisfying the core system functional requirements. Such concepts in turn were integrated into software architectural design language descriptions and configuration management tools (Narayanaswamy & Scacchi, 1987). However, reliance on software architecture descriptions is not readily found in either conventional game or mod development. Hentonnen, Matinlassi, Niemela, and Kanstren (2007) examine how software plug-ins support architectural extension, while Leveque, Estublier, and Vega (2009) investigate how extension mechanisms like views and model-based systems support extension also at the architectural level. Last, the modern Web architecture is itself designed according to principles of extensibility through open interfaces, migration across software versions, network data content/hypertext transfer protocols, and representational state transfer (Fielding & Taylor, 2002). Mod-friendly networked multi-player games appear to take advantage of these capabilities.

Elsewhere, Batory, Johnson, MacDonald, and von Heeder (2002) describe how domain-specific (scripting) languages and software product lines provide support software extension, and it now seems clear that such techniques are commonly used in games that are open for modding. Finally, FOSSD has become another approach to extensible software engineering in practice (Scacchi, 2007).

Therefore, software extensions and extensibility is a foundational concept in software engineering, and thus to no surprise, also foundational to the development of game mods. However, the logical connections and common/uncommon legacy remain under specified, which this paper seeks to address and update.

C. Four Types of Game Mods

1. User Interface Customizations

User interfaces to games embody the practice and experience of interfacing users (game players) to the game system and play experience designed by game developers. Game developers act to constrain and govern what users can do, and what kinds of experiences they can realize. Some users in turn seek to achieve some competitive advantage during game play by modding the user interface software for their game, when so enabled by game developers, to acquire or reveal additional information that the users believe will help their play performance and experience. User interface add-ons subsequently act as the medium through which game development studios support game product customization as a strategy for increasing the likelihood of product success through end-user satisfaction (Burnett et al., 2004)

Three kinds of user interface customizations can be observed. First and most common, is the player's ability to select, attire, or accessorize a *player's in-game identity*. Second, is for players to customize *the color palette and representational framing borders* of the their game display within the human-computer interface, much like what can also be done with Web browsers and other end-user software applications. Third, are *user interface add-on modules* that modify the player's in-

game information management dashboard and that do not modify game play rules or functions. These add-ons provide additional information about game play or game state that may enhance the game play experience, as well as increase a player's sense of immersion or omniscience within the game world through sensory or perceptual expansion. This in turn enables awareness of game events not visible in the player's current in-game view. Consequently, the first two kinds of customizations result from meta-data selections within parametric system functions, whereas the third represents a traditional kind of modular extension that does not affect the pre-existing game's functional requirements.

2. Game Conversions

Game conversion mods are perhaps the most common form of game mods. Most such conversions are partial, in that they add or modify (a) in-game characters including user-controlled character appearance or capabilities, opponent bots, cheat bots, and non-player characters, (b) play objects like weapons, potions, spells, and other resources, (c) play levels, zones, maps, terrains, or landscapes, (d) game rules, or (e) play mechanics. Some more ambitious modders go as far as to accomplish (f) total conversions that create entirely new games from existing games of a kind that are not easily determined from the originating game. For example, one of the most widely distributed and played total game conversions is the *Counter-Strike* (CS) mod of the *Half-Life* (HL) first-person action game from Valve Software. The success of the CS mod gave rise to millions of players preferring to play the mod over the original HL game, then other modders began to access the CS mod to further convert in part or full. Valve Software subsequently modified its game development and distribution business model to embrace game modding as part of the game play experience that is available to players who acquire a licensed copy of the HL product family. Valve has since marketed a number of CS variants that have sold over 10 million copies as of 2008, thus denoting the most successful game conversion mod, as well as the most lucrative in terms of subsequent retail sales derived from a game mod.

Another example is found in games converted to serve a purpose other than entertainment, such as the development and use of games for science, technology, and engineering applications. For instance, the *FabLab* game (Scacchi, 2010) is a conversion of the *Unreal Tournament 2007* retail game from a first-person action shooter to a simulator for training semiconductor manufacturing technicians in diagnosing and treating potentially hazardous materials spills in a cleanroom environment. However, this conversion is not readily anticipated by knowledge of the Unreal games or underlying game engine, though it maintains operational compatibility with the Unreal game itself. Thus, game conversions can repurpose the look, feel, and intent of a game across application domains, while maintaining a common software product line (cf. Batory et al., 2002).

Finally, it is common practice that the underlying game engine has one set of license terms and conditions to protect original work (e.g., no redistribution), whereas game mods can have a different set of terms and conditions from a derived work (e.g., redistribution allowed only for a game mod, but not for sale). In this regard, software licenses embody the business model that the game development studio or publisher seeks to embrace rather than just a set of property rights and constraints. For example, in *Aion*, an MMOG from the South Korean game studio NCSoft, no user created mods or user interface add-ons are allowed. Attempting to incorporate such changes would therefore conflict with its end-user license agreements (EULA) and subsequently put such user-modders at risk of losing their access to networked *Aion* multi-player game play. In contrast, the MMOG *World of Warcraft* (WoW) allows for UI customization mods and add-ons only, but no other game conversions, no reverse engineering game engine, and no activity intended to bypass WoW's encryption mechanisms. And, in one more variation, for games like *Unreal Tournament*, *Half-Life*, *NeverWinterNights*, *Civilization* and many others, the EULAs encourage modding and the free redistribution of mods without fee to others who must have a licensed game copy, but do not encourage reverse engineering or redistribution of the game engine required to run the mods. This restriction in turn helps game companies realize the benefit of increased game sales by players who

want to play with known mods, rather than with the unmodded game as sold at retail. Thus, mods help improve game software sales, revenue, and profits for the game development studio, publisher, and retailer.

3. Machinima

Machinima can be viewed as the product of modding efforts that intend to modify the visual replay of game usage sessions. Machinima employ computer games as their creative media, such that these new media are mobilized for some other purpose (e.g., creating online cinema or interactive art exhibitions). Machinima focuses attention to playing and replaying a game for the purpose of story telling, movie making, or retelling of a daunting or high efficiency game play/usage experience (Marino, 2004). Machinima is a form of modding the experience of playing a specific game through a recording of its visual play session history so as to achieve some other ends beyond the enjoyment (or frustration) of game play. These play-session histories can then be further modded via video editing or remixing with other media (e.g., audio recordings) to better enable cinematic storytelling or creative performance documentation. Machinima is thus a kind of play/usage history process re-enactment (cf. Scacchi, 1998) whose purpose may be documentary (replaying what the player saw or experienced during a play session) or cinematic (creatively steering a play session so as to manifest observable play process enactments that can be edited and remixed off-line to visually tell a story). Thus, machinima mods are a kind of extension that is not bound to the architecture of the underlying game system, except for how the game facilitates a user's ability to structure and manipulate emergent game play to realize a desired play process enactment history.

4. Hacking Closed Game Systems

Hacking a closed game system is a practice whose purpose oftentimes seems to be in direct challenge to the authority of commercial game developers that represent large, global corporate interests. Hacking proprietary game software is

often focused not so much on how to improve competitive advantage in multi-player game play, but instead is focused on expanding the range of experiences that users may encounter through use of alternative technologies (Huang, 2003; Scacchi, 2004). For example, Huang's (2003) study instructs readers in the practice of "reverse engineering" as a strategy to understand both how a game platform was designed and how it operates in fine detail, as a basis for developing new innovative modifications or original platform designs, such as installing and running a Linux open source operating system (instead of Microsoft's proprietary closed source offering). Although many game developers seek to protect their intellectual property (IP) from reverse engineering through EULA whose terms attempt to prohibit such action under threat of legal action, reverse engineering is not legally prohibited nor discouraged by the courts. Consequently, the practice of modding closed game systems is often less focused on enabling players to achieve competitive advantage when playing retail computer games, but instead may encourage those few so inclined for how to understand and ultimately create computing innovations through reverse engineering or other DIY game system modifications. Thus, closed game system modding is a style of software extension by game modders who are willing to forego the "protections" and quality assurances that closed game system developers provide, in order to experience the liberty, skill, and knowledge acquisition, as well as the potential to innovate, that mastery of reverse engineering affords. Consequently, players/modders who are willing to take responsibility for their actions (and not seek to defraud game developers or publishers due to false product failure warranty claims or copyright infringement) can enjoy the freedom to learn how their gaming systems work in intimate detail and potentially learn about game system innovation through discovery and reinvention with the support of others who are like-minded (cf. Scacchi, 2007).

Finally, games are one of the most commonly modified types of software that are transformed into "pirated games" that are "illegally downloaded." Such game modding practice is focused on engaging a kind of meta-game that involves modding game IP from closed to (more) open. Thus, game piracy has become

recognized as a collective, decentralized, and placeless endeavor (i.e., not a physical organization) that relies on torrent servers as its underground distribution venue for pirated game software. As recent surveys of torrent-based downloads reveal, in 2008 the top 10 pirated games represented about 9 million downloads, while in 2009 the top 5 pirated games represented more than 13 million downloads, and in 2010 the top 5 pirated games approached 20 million, all suggesting a substantial growth in interest in and access to such modded game products. Thus, we should not be surprised by the recent efforts by game system hackers that continue to demonstrate the vulnerabilities of different hardware and software-based techniques to encrypt and secure closed game systems from would be hackers. However, it is also very instructive to learn from these exploits how difficult it is to engineer truly secure software systems, whether such systems are games or some other type of application or package.

D. Game Modding Software tools and Support

Games are most often modded with tools that provide access to an unencrypted representation of the game software or game platform. Such a representation is accessed and extended via a domain-specific (scripting) language. Although it might seem the case that game vendors would seek to discourage users from acquiring such tools, we observe a widespread contrary pattern.

Game system developers are increasingly offering software tools for modifying the games they create or distribute as a way to increase game sales and market share. Game/domain-specific Software Development Kits (SDKs) provided to users by game development studios represent a contemporary business strategy for engaging users to help lead product innovation from outside the studio. Once Id Software, maker of the *DOOM* and *Quake* game software product line, and also Epic Games, maker of the *Unreal* software game product line, started to provide prospective game players/modders with software tools that would allow them to edit game content, play mechanics, rules, or other functionality, other competing game development studios were pressured to make similar offerings or face a possible

competitive disadvantage in the marketplace. However, these tools do not provide access to the underlying source code that embodies the proprietary game engine—a large software program infrastructure that coordinates computer graphics, user interface controls, networking, game audio, access to middleware libraries for game physics, and so forth. However, the complexity and capabilities of such a tool suite mean that any one, or better said, any game development or modding team, can now access modding tools or SDKs to build commercial-quality games. However, mastering these tools appears to be a significant undertaking likely to be of interest only to highly committed, would-be game developers who are self-supported or self-organized.

In contrast to game modding platforms provided by game development studios, there are also alternatives provided by the end-user community. One approach can be seen with facilities provided in *Garry's Mod* mod-making package that you can use to construct a variety of fanciful contraptions as user created art works, or to create comic books, program game conversions, and produce other kinds of user created content. However, this package requires that you own a licensed game like *Counter-Strike: Source*, *Half-Life2* or *Day of Defeat: Source* from Valve Software.

A different approach to end-user game development platforms can be found arising from free/open source software games and game engines. The *DOOM* and *Quake* games and game engines were released as free software subject to the GPL once they were seen by Id Software as having reached the end of their retail product cycle. Hundreds of games/engines have been developed and released for download starting from the free/open source software that was the platform of the original games. However, the content assets for many of these games (e.g., in-game artwork) are not covered by the GPL, and so user-developers must still acquire a licensed copy of the original game if its content is to be reused in some way.

Nonetheless, some variants of the user-created GPL'd games now feature their own content that is limited/protected by Creative Commons licenses.

E. Opportunities for Modding and Software Engineering

Game modding demonstrates the practical value of software extension as a user-friendly approach to custom software. Such software can extend games open to modding into diverse product lines that flourish through reliance on domain-specific game scripting languages and integrated software development kits. Modding also demonstrates the success of end-users learning how to extend software to create custom user interface add-ons, system conversions, replayable system usage documentaries and movies, as well as to discover security vulnerabilities. Therefore, game modding represents a viable form of end-user engineering of complex software that may be transferable to other domains.

Acknowledgements

The research described in this paper has been supported by grants #0808783 and #1041918 from the U.S. National Science Foundation and grants #N00244-10-1-0064 and #N00244-10-1-0077 from the Acquisition Research Program at the Naval Postgraduate School. No review, approval, or endorsement is implied.

References

Batory., D., Johnson, C., MacDonald, B., & von Heeder, D. (2002). Achieving extensibility through product lines and domain specific languages: A case study. *ACM Trans. Software Engineering and Methodology*, *11*(2), 191–214.

Burnett, M., Cook, C., & Rothermel, G. (2004). End-user software engineering. *Communications ACM*, *47*(9), 53–58.

El-Nasr, M. S., & Smith, B. K. (n.d.). Learning through game modding. *ACM Computers in Entertainment*, *4*(1). Article 3B.

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

Huang, A. (2003). *Hacking the Xbox: An introduction to reverse engineering.* San Francisco, CA: No Starch Press.

Henttonen, K., Matinlassi, M., Niemela, E., & Kanstren, T. (2007). Integrability and extensibility evaluation in software architectural models—A case study. *The Open Software Engineering Journal, 1*(1), 1–20.

Narayanaswamy, K., & Scacchi, W. (1987). Maintaining evolving configurations of large software systems. *IEEE Trans. Software Engineering*, *SE-13*(3), 324–334.

Leveque, T., Estublier, J., & Vega, G. (2009). Extensibility and modularity for model-driven engineering environments. In *Proceedings of the 16th IEEE Conference on Engineering Computer-Based Systems* (*ECBS 2009*; pp. 305–314).

Marino, P. (2004). *3D game-based filmmaking: The art of machinima*. Scottsdale, AZ: Paraglyph Press.

Parnas, D. L. (1979). Designing software for ease of extension and contraction. *IEEE Trans. Software Engineering*, *SE-5*(2), 128–138.

Scacchi, W. (1998). Modeling, integrating, and enacting complex organizational processes. In K. Carley, L. Gasser, & M. Prietula (Eds.), *Simulating organizations: Computational models of institutions and groups* (pp. 153–168). Cambridge, MA: MIT Press.

Scacchi, W. (2002, February). Understanding the requirements for developing open source software. *IEE Proceedings—Software Engineering, 149*(1), 24–39. Revised version in K, Lyytinen, P. Loucopoulos, J. Mylopoulos, and W. Robinson (Eds.), *Design Requirements Engineering: A Ten-Year Perspective*, LNBIP 14, Springer-Verlag, 467-494, 2009.

Scacchi, W. (2004, January/February). Free/open source software development practices in the game community. *IEEE Software*, *21*(1), 59–67.

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

Scacchi, W. (2010). Game-based virtual worlds as decentralized virtual activity systems. In W. S. Bainbridge (Ed.), *Online worlds: Convergence of the real and the virtual* (pp. 225–236). New York: Springer.

THIS PAGE INTENTIONALLY LEFT BLANK

V. Final Report Discussion and Prospects for Future Acquisition Research

Walt Scacchi and Thomas A. Alspaugh
Institute for Software Research.
University of California, Irvine USA

A. Overview

Overall, our efforts developed in this research project and described in this report sought to articulate the acquisition research problem with respect to the issues identified above in order to determine what types or kinds of answers can be realized through this investigation. Subsequently, our efforts focused on the following four activities:

- Investigating the interactions between software system acquisition guidelines, software system requirements, requirements for OSS, and consequences of alternative software system architectures that incorporate different mixes of OSS components, SPLs with open APIs and open standards (Scacchi & Alspaugh, 2008; Alspaugh, Asuncion, & Scacchi, 2009a, 2009b, 2009c; Scacchi, Alspaugh, & Asuncion, 2010). This entails exploring the balance between development, verification, and validation of property and security rights, as well as contractual obligations within continuously improving OSS system elements while managing the evolution of OA systems at design-time, build-time, and release and run-time.
- Developing and refining the formal foundations for establishing acquisition guidelines for use by program managers seeking to provide software-intensive systems in cost reducing ways that rely on development and deployment of secure OA systems using OSS and SPL technology and processes (Alspaugh et al., 2009c).
- Developing concepts for the design of a comprehensive automated system that can support acquisition of OA systems so as to determine their conformance to acquisition guidelines/policies, contracts, and related license management issues (Alspaugh et al., 2009a; Asuncion, 2009).

- Documenting and presenting final results (Scacchi & Alspaugh, 2011) at the *8th Annual Acquisition Research Conference*, in Monterey, CA, May 2011, as well as at related research venues and publications where we can elicit the strongest critical feedback on our research efforts and results.

B. Inter-Project Research Coordination

We continue to believe that we are extremely well positioned to continue to leverage our recent research work and results (Scacchi & Alspaugh, 2008; Alspaugh et al., 2009a, 2009b, 2009c; Scacchi et al., 2010) with the effort described here. We continued to build on our recent research efforts in OSS (Scacchi, 2007, 2011a, 2011b) and software requirements-architecture interactions (Asuncion, 2009; Scacchi, 2009; Scacchi & Alspaugh, 2008), as well as our track record in prior acquisition research studies. Similarly, we find current related research supported by the Department of Defense (DoD) addressing related issues in OSS (Hissam, Weinstock, & Bass, 2010) also influences our proposed effort. In addition, our effort builds from and contributes to research on software system acquisition within the DoD, whether focusing on SPLs (Bergey & Jones, 2010; Guertin & Clements, 2010), or on how to improve software system acquisition through workforce upgrades and government-industry teaming (Heil, 2010). Thus, we believe that our complementary research places us at an extraordinary advantage to conduct the proposed study that addresses a major strategic acquisition goal of the DoD and the three military Services (Starrett, 2007; Weathersby, 2007; Wheeler 2007).

C. Prospects for Longer Term Acquisition-Related Research

Each of the military Services has committed to orienting their major system acquisition programs around the adoption of an OA strategy that in turn embraces and encourages the adoption, development, use, and evolution of OSS. Thus, it would seem there is a significant need for sustained research that investigates the interplay and inter-relationships between (a) current/emerging guidelines for the acquisition of software-intensive systems within the DoD community (including contract management and software development issues) and (b) how software

systems that employ an OA incorporating OSS products and production processes are essential to improving the effectiveness of future, software-intensive program acquisition efforts. Consequently, we have focused our research project, and the results appearing in this final report, to continue to lay new foundations for long-term acquisition-related research in support of the Acquisition Research Program based at the Naval Postgraduate School.

Acknowledgments

The research described in this Final Report is supported by grant #N00244-10-1-0077 from the Acquisition Research Program at the Naval Postgraduate School.

References

Alspaugh, T. A, Asuncion, H., & Scacchi, W. (2009a, May). Analyzing software licenses in open architecture software systems. *Workshop on Emerging Trends in FLOSS Research and Development, International Conference on Software Engineering*, Vancouver, Canada.

Alspaugh, T. A, Asuncion, H., & Scacchi, W. (2009b, May). Software licenses, open source components, and open architectures. In *Proceedings of the 6th Annual Acquisition Research Symposium* (NPS-AM-09-026). Monterey, CA: Naval Postgraduate School.

Alspaugh, T. A, Asuncion, H., & Scacchi, W. (2009c, September). Intellectual property rights requirements for heterogeneously licensed systems. In *Proceedings of the 17th International Conference on Requirements Engineering (RE09*; pp. 24–33). Atlanta, GA.

Alspaugh, T. A., Asuncion, H. & Scacchi, W. (2011, July). Presenting software license conflicts through argumentation. *22nd International Conference on Software Engineering and Knowledge Engineering* (SEKE2011), Miami, FL.

Bergey, J., & Jones, L. (2010). Exploring acquisition strategies for adopting a software product line approach. In *Proceedings of the 7th Annual Acquisition Research Symposium* (Vol. 1, pp. 111–122). Monterey, CA: Naval Postgraduate School.

Guertin, N. & Clements, P. (2010). Comparing acquisition strategies: Open architecture versus product lines. In *Proceedings of the 7th Annual Acquisition*

Research Symposium (Vol. 1, pp. 78–90). Monterey, CA: Naval Postgraduate School.

Heil, J. (2010). Enabling software acquisition improvement: Government and industry software development team acquisition model. In *Proceedings of the 7th Annual Acquisition Research Symposium* (Vol. 1, pp. 203–218). Monterey, CA: Naval Postgraduate School.

Hissam, S., Weinstock, C. B., & Bass, L. (2010). On open and collaborative software development in the DoD. In *Proceedings of the 7th Annual Acquisition Research Symposium* (Vol. 1, pp. 219–235). Monterey, CA: Naval Postgraduate School.

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

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

Scacchi, W. (2011a, May). Modding as a basis for developing game systems. *1St Workshop Games and Software Engineering* (GAS'11), *33rd International Conference on Software Engineering*, Waikiki, Honolulu, HI.

Scacchi, W. (2011b, October). Modding as an open source approach to extending computer game systems. In S. Hissam, B. Russo, M. G. de Mendonca Neto, & F. Kan (Eds.), *Open source systems: Grounding research.* In *Proceedings of the 7Th IFIP International Conference on Open Source Systems* (pp. 62–74). IFIP ACIT 365, (Best Paper award), Salvador, Brazil.

Scacchi, W., & Alspaugh, T. (2008, May). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proceedings of the 5th Annual Acquisition Research Symposium* (NPS-AM-08-036). Monterey, CA: Naval Postgraduate School.

Scacchi, W., & Alspaugh, T. (2011, May). Advances in the acquisition of secure systems based on open architectures. *8Th Annual Acquisition Research Symposium*, Monterey, CA, Naval Postgraduate School.

Scacchi, W., Alspaugh, T., & Asuncion, H. (2010, May). The challenge of heterogeneously licensed systems in open architecture software ecosystems. In *Proceedings of the 7th Annual Acquisition Research Symposium* (Vol. 1, pp. 91–110). Monterey, CA: Naval Postgraduate School.

Starrett, E. (2007, May). Software acquisition in the Army. *Crosstalk: The Journal of Defense Software Engineering*, *48*. Retrieved from http://stsc.hill.af.mil/crosstalk

Weathersby, J. M. (2007, June). 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.

Wheeler, D. A. (2007, June). Open source software (OSS) in U.S. Government acquisitions. *The DoD Software Tech News*, *10*(2), 7–13.

THIS PAGE INTENTIONALLY LEFT BLANK

2003 - 2011 Sponsored Research Topics

Acquisition Management

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

Contract Management

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

Financial Management

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

Human Resources

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

Logistics Management

- Analysis of LAV Depot Maintenance
- Army LOG MOD
- ASDS Product Support Analysis
- Cold-chain Logistics
- Contractors Supporting Military Operations
- Diffusion/Variability on Vendor Performance Evaluation
- Evolutionary Acquisition
- Lean Six Sigma to Reduce Costs and Improve Readiness

- Naval Aviation Maintenance and Process Improvement (2)
- Optimizing CIWS Lifecycle Support (LCS)
- Outsourcing the Pearl Harbor MK-48 Intermediate Maintenance Activity
- Pallet Management System
- PBL (4)
- Privatization-NOSL/NAWCI
- RFID (6)
- Risk Analysis for Performance-based Logistics
- R-TOC AEGIS Microwave Power Tubes
- Sense-and-Respond Logistics Network
- Strategic Sourcing

Program Management

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

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

THIS PAGE INTENTIONALLY LEFT BLANK

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

www.acquisitionresearch.org