ISSUES IN DEVELOPMENT AND MAINTENANCE OF OPEN ARCHITECTURE SOFTWARE SYSTEMS

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

Revised version to appear in

CROSSTALK: The Defense Software Engineering Journal (2017)

ABSTRACT: This article identifies and describes a set of six emerging issues that affect the engineering of open architecture software systems that integrate proprietary and open source software components. These interdependent issues identify problems for software engineering research and practice associated with: (a) unknown or unclear open architecture software representations; (b) systems subject to heterogeneous software licenses; (c) cybersecurity of open architecture software systems; (d) build, release, and deployment processes and process automation; (e) evolution practices for open architecture software; and (f) new business models affecting the acquisition costs of open architecture software components.

INTRODUCTION

There is growing interest within the Defense Community in transitioning to acquiring complex, cybersecure open architecture (OA) software systems within an agile and adaptive ecosystem [8,10,11,14,15]. In such a world, proprietary closed source (CSS) or open source software (OSS) components may be acquired from alternative software producers or system integrators across the system life cycle. This is envisioned to enable more competition and ideally lower costs and increase the quality of software elements that arise from a competitive marketplace [8]. But this adaptive agility to mix, match, reuse, mashup, swap, or reconfigure integrated systems or components requires that systems be compatible with, or designed to utilize, an OA—a software representation that identifies component types, component interconnections, and open APIs [14].

The common core of cybersecure OA command and control (C2) systems resembles many enterprise business systems, as C2 are a kind of management information system for navigating, mapping, tracking resources; scheduling people and other resources; producing plans and documentation; supporting online email, voice or video communications. Figure 1 depicts an OA representation for such a kind of system. This OA representation can be read as a “reference model” for a C2 software product line. But there are an emerging set of software engineering research and practice issues that arise and affect the development of new C2 systems, especially when CSS and OSS components are combined into an integrated system.

We identify six kinds of emerging software life cycle challenges that we have observed within the U.S. Defense Community as they have moved to OA systems for C2 that integrate, operate and maintain contemporary OSS and CSS components. These challenges follow from our ongoing efforts within the

Defense Community that build on and refine our previous efforts in this area [14,15], and point to areas of practice requiring further software engineering (SE) research.

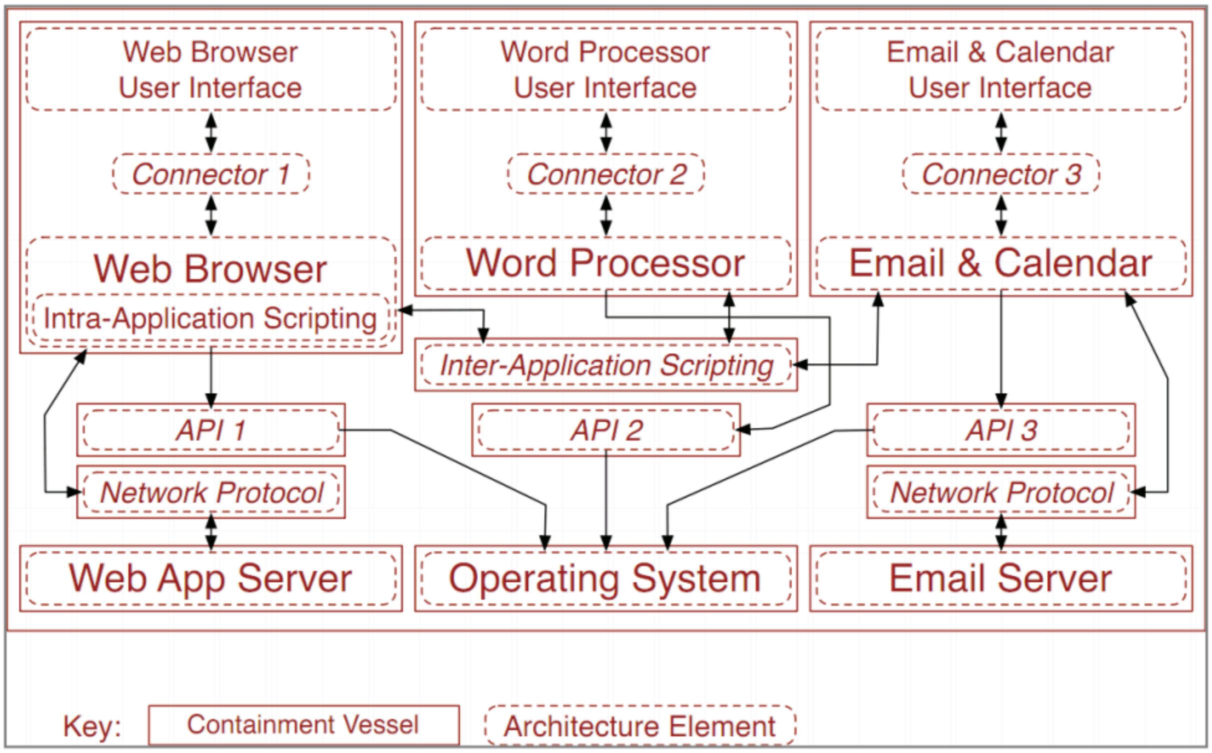

Figure 1. An OA reference model for common software component types interconnected within an integrated, cybersecure C2 systems [14].

Unknown or unclear OA system representations

This first kind of challenge arises when acquiring new or retrofitting legacy software systems that lack an open or explicit architectural representation. Such a representation should identify and model major software system components, interfaces, interconnections and remote services (if any). Though OA reference models are in use within the SE research community, contemporary C2 generally lack such descriptions or representations that are open, sharable, or reusable. This may be the result of legacy contractor business practices that see software architectures as proprietary intellectual property (IP), even when OSS components are included, or when applications sub-systems are entirely made of OSS code. An alternative explanation reveals that complex software systems like common Web browsers (*Mozilla Firefox, Google Chrome, Apple Safari, Microsoft Edge*) have complex architectures that integrate millions of SLOC that are not well understood, and that entail dozens of independently-developed software elements with complex APIs and IP licenses that shift across versions [12]. This implies the effort to produce an explicit OA reference model is itself a daunting task of architectural discovery, restructuring, and continuous software evolution [5,7]. Thus, new ways and means for extracting software components, interconnections, and interfaces and transforming them into higher-level architectural representations are needed.

Heterogeneously licensed OA systems

OSS components are subject to widely varying copyright licenses, end-user license agreements, digital civil rights, and other IP protections. The Open Source Institute recognizes dozens of OSS licenses that are in use, though the top 10 represent more than 90% of the open source ecosystem [9]. This is especially true for OSS components or application systems that incorporate source code from multiple, independent OSS development projects, such as found in contemporary Web browsers like Firefox and Chrome that incorporate components from dozens of OSS projects, most with diverse licenses [12]. This means that OSS application systems are subject to complex software IP obligations and rights that may defy tracking, or entail contradictory obligations/rights [1]. Determining overall IP obligations for such systems is generally beyond the scope of expertise for software developers, as well as most corporate lawyers. Furthermore, we have observed many ways in which IP licenses interact within an OA software system, such that different architectural design choices that configure a fixed set of software components results in different overall system obligations and rights. Understanding multiple license interaction and IP mis-matches is far too confusing for most people and a source of legal expense, or alternatively expensive indemnification insurance policies by the software producers or system integrators. Nonetheless, in our view, OA software ecosystems are defined, delimited, and populated with *niches* that locate specific integrated system solutions [12]. Furthermore, we see that these niches effectively have *virtual IP licenses* that must be calculated via the obligations and rights that propagated across integrated system component licenses via union, intersection, and subsumption relations among them [4]. Such calculation is daunting, and begs for a simpler, tractable, and computationally enforced scheme that can scale to large systems composed from many components. In such a scheme, OSS/CSS licenses could formalize IP obligations as operational requirements (i.e., computationally enforceable), at the integrated system level and instantiated by system integration architects. Similarly, customer/user rights are then non-functional requirements that can be realized and validated as access/update capabilities propagated across the integrated system [3].

Cybersecurity for OA systems

Cybersecurity is a high priority requirement in all C2 systems, applications, and platforms [14]. No longer is cybersecurity something to be addressed after C2 are developed and deployed—cybersecurity must be considered throughout the design, development, deployment, and evolution of C2. However, the best ways and means for addressing cybersecurity requirements are unclear, and oftentimes somewhat at odds with one another depending on whether cybersecurity capability designs are specific to the: C2 platform (e.g., operating system or processor virtualization; utilization of low-level operating system access control or capability mechanisms); component producer (secure programming practices and verification testing); system integrator (e.g., via the use of secure data communications protocols and data encryption); customer deployment setting (mobile: air-borne or ship-board; fixed: offices, briefing rooms, operations centers); end-user authentication mechanisms; or acquisition policy (e.g, reliance on third-party audit, certification, assurance of system cybersecurity). However, in reviewing these different arenas for cybersecurity, we have found that the cybersecurity requirements or capabilities can be expressed in much the same way as IP licenses: using concise, testable formal expressions of obligations and rights. Some suggestive examples follow (capital letters are placeholders that denote specified system, service, or component contexts).

- The obligation for a user to verify her authority to invoke application software and access data in compartment T, by password check or other specified authentication process.

- The obligation for all components connected to specified component C to grant it the capability to read and update data in compartment T.
- The obligation to reconfigure a system in response to detected threats, when given the right to select and include different component versions, or executable component variants.
- The right to read and update data in compartment T using the licensed component C.
- The right to replace component C with a certified secure component D.

These examples show how cybersecurity requirements can be expressed or paraphrased into/from restricted natural language (e.g., using a domain-specific language) into composite specifications that denote "security licenses" [1,2,14]. In this way, it should be possible to develop new software analysis tools whose purpose is to interpret cybersecurity obligations as operational constraints (executable) or provided capabilities (access control or update privileges), through mechanisms analogous to those used for analyzing software licenses [1,4,14], and determine how component or sub-system-specific obligations and rights can be propagated across a system's architecture. Consequently, we believe that cybersecurity can therefore in the future be addressed using explicit, computational OA representations that are attributed with both IP and cybersecurity obligations and rights.

Build, Release, and Deployment (BRD) Processes and Process Automation

C2 applications represent complex software systems that are often challenging to produce and maintain especially when initially conceived as bespoke systems. To no surprise, acquisition of these systems requires a development life cycle approach, though some system elements may be fully-formed components that are operational as packaged software (e.g., commercial database management systems, Web browsers, Web servers, user interface development kits/frameworks). C2 system development is infrequently clean-sheet and even less likely to be so in the future. As a result, component-based system development approaches are expected to dominate. This implies system integrators (or even end-users) must perform any residual source code development, inter-app integration scripting, or intra-app extension script development. But software process challenges arise along the way [13].

First, as noted earlier, the issue of whether there is an explicit, open source OA design representation, preferably one that is not just a diagram, but instead is expressed in a computational architectural design language. With only a diagram or less, there is little or no guidance for how to determine whether/how an operational software implementation is verifiable or compliant with its OA requirements or acquisition policies. Current acquisition policy guidance calls for provision or utilization of standardized, open APIs intended to increase software reuse, selection of components from alternative producers, and post-deployment system extensions [8].

Second, there is the issue arising from system development practices based on utilization of software components, integrated sub-systems, or turn-key application packages. These software elements come with their own, possibly unknown requirements that are nonetheless believed to exist and be knowable with additional effort [3]. They also come with either OSS code or CSS executables, along with their respective APIs. These components must be configured to align with the OA specification. Consequently, software tool chains or workflow automation pipelines are utilized to build and package internal/external executable, version-controlled operational software releases. We have found many diverse automated software process pipelines are used across and sometimes within software integration activities [13]. These pipelines take in OSS code files, dependent libraries, or repositories (e.g., *GitHub*) and build executable version instances that are then subjected to automated testing regimes ranging from simple "smoke tests" to extensive regression testing. Successful builds eventually turn into packaged releases that may or may not be externally distributed and deployed as ready-to-install executables. While this all seems modest and tractable, when one sees the dozens of different OSS tools used in different combinations across different target platforms, it becomes clear that what is simple in the small, becomes a complex operations and maintenance activity when the scale of deployment increases.

Another complication that is now beginning to be recognized within and across BRD processes and process automation pipelines arises in determining when and how different BRD tool chain versions/configurations can mediate cybersecurity requirements in the target system being built or maintained. For instance, many software system builds and deployed releases are assumed to integrate to functionally equivalent CSS components. However, many CSS components are not included in distribution releases due to IP restrictions, and thus must be linked and configured into already deployed operational systems. We have also observed and reported how functionally equivalent variants as well as functionally similar versions may or may not be produced by BRD tool chains, either by choice or by unintentional consequence. This in our opinion gives rise for the need for explicit open source models of BRD process automation pipelines that can be analyzed, reused, and shared, as well as systematically tested to determine whether release versions/variants can be verified and/or validated to produce equivalent or similar releases that preserve prior cybersecurity obligations and usage rights.

Software Evolution Practices Transmitted Across the OA Ecosystem

Software evolution is among the most-studied of SE processes. While often labeled as "software maintenance," a frequently profitable activity mediated through maintenance contracts from software producers to customers, the world of OSS development projects and practices suggest a transition to a world of c*ontinuous software development*—one that foreshadows the emergence of continuous SE processes or software life cycles that just keep cycling until interest in the software falters or spins off into other projects. OSS development projects rely on OSS tools that themselves are subject to ongoing development, improvement, and extension, as are the software platforms, libraries, code-sharing repositories, and end-user applications utilized by OSS developers to support their development work. Developers entering, progressing, or migrating within/across OSS projects further diversifies the continuous development of the most successful and widely used OSS components/apps. This dynamism in turn produces many ways for OSS systems, or OA systems that incorporate OSS components, to evolve.

Figure 2 portrays different software evolution patterns, paths, and practices we have observed arising with new C2 applications [12]. Here we see paths from a currently deployed, operational software system release, to a new deployed release update—something most of us now accept as routine as software updates are propagated across the Internet from producers, through integrators, to customers and end-users.

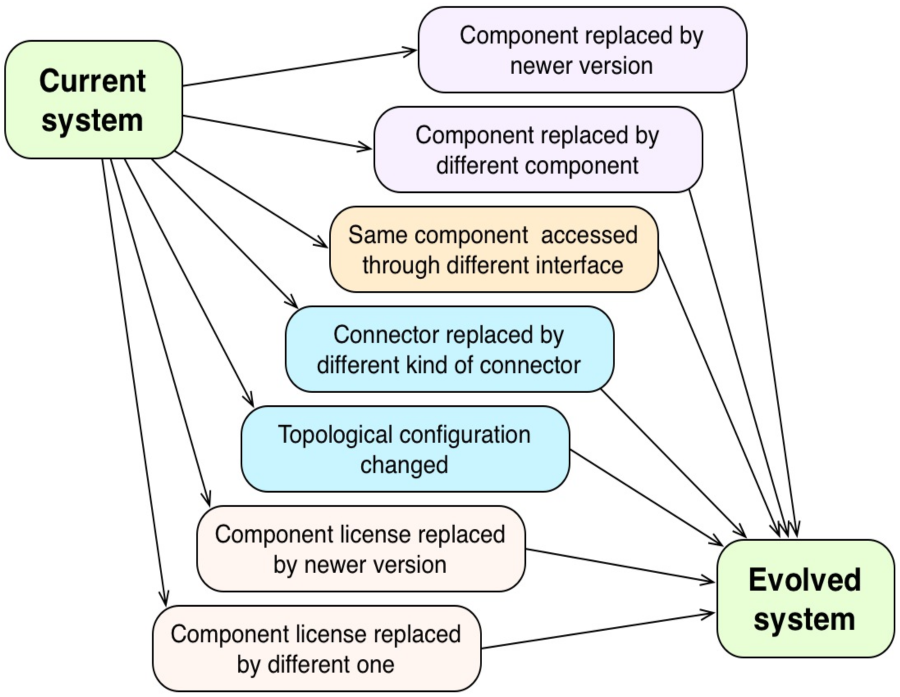
Current system
Component replaced by newer version
Component replaced by different component
Same component accessed through different interface
Connector replaced by different kind of connector
Topological configuration changed
Component license replaced by newer version
Component license replaced by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

Figure 2. Different paths and mechanisms through which currently operational OA software system components can evolve [12].

Integrated OA systems can evolve through upgrades of *functionally equivalent* component variants (patches), as well as through substitution of *functionally similar* software components sourced from other producers or integrators. In Figure 3, we show a generic situation that entails identifying how an OA consistent with that depicted in Figure 1 may accommodate the substitution and replacement of a locally installed word processor application program (like *Microsoft Word*) with a remote Web-based word processing software services (for example, *Google Docs* or *Microsoft Office 365*). Also note how the software IP licenses compose across the four similar release instances shown.

This capability is a result of utilizing an OA that constitutes a reference model aligned with a vendor-neutral software product line. This is also a capability sought by customer organizations, and sometimes encouraged by software producers to accommodate their evolving business models (discussed below). While the OA remains constant, the location of the component has moved from local to remote/virtual, as has its evolutionary path. Similarly, the propagation of IP and cybersecurity requirements of the local versus remote component has changed in ways that are unclear, and entail a different, evolved assurance scheme.

Overall, the evolution of software components, component licenses, component interconnects and interconnections, and interconnected component configurations are now issues that call for SE research efforts to help make such patterns, paths, and practices more transparent, tractable, manageable, and scalable within an OA software ecosystem, as well as for customer organizations that seek the benefits of openness, sharing, and reuse.

New Business Models for Ongoing Acquisition of Software Components and Apps

The last issue we address is the newest in this set of six for consideration for new SE research. While SE research and practice has long paid attention to software economics, the challenges of software cost estimation are evolving in light of new business models being put into practice by software producers and system integrators.

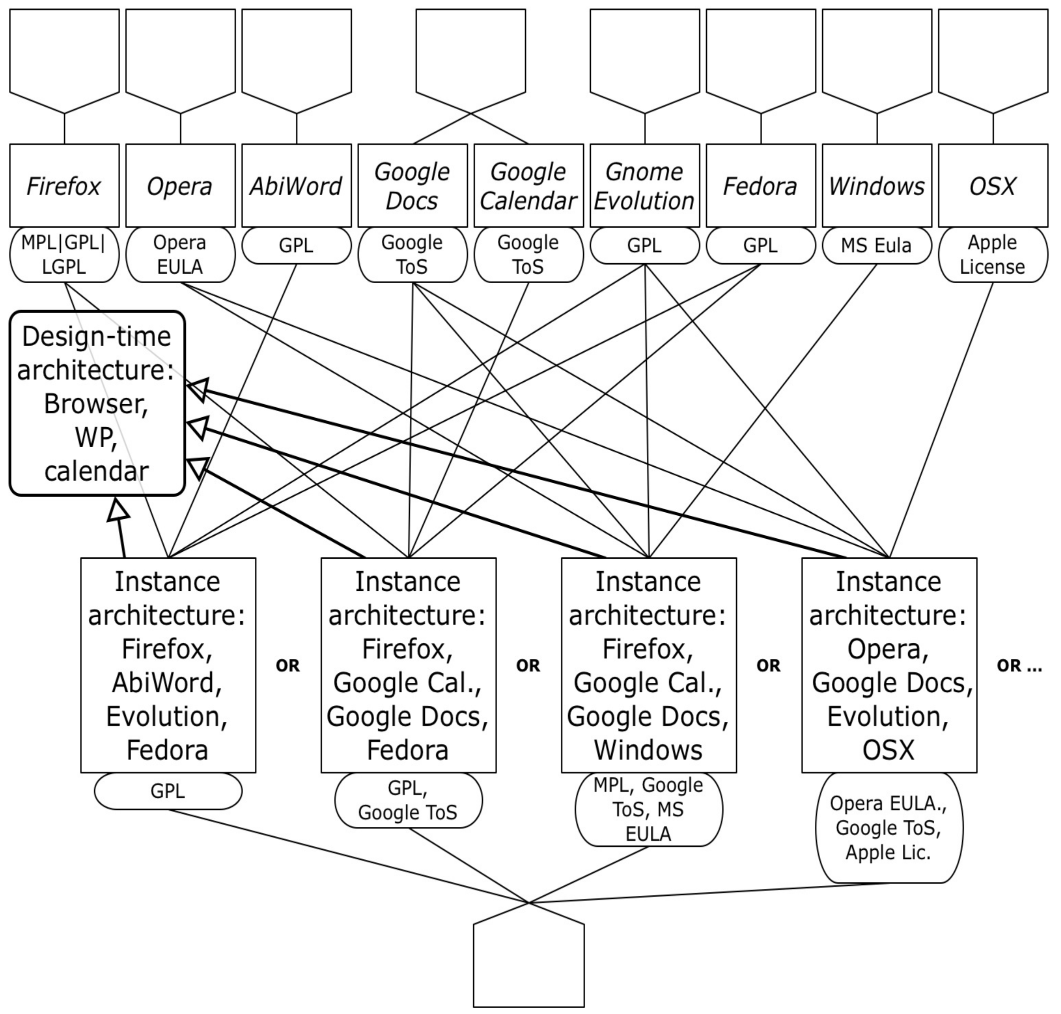
Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
MPL|GPL| LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
OR
Instance architecture: Firefox,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

Figure 3. Alternative configurations of integrated instance releases of components and IP licenses consistent with the OA in Figure 1 that are treated as functionally similar by customer organizations.

In the past, system development projects were often managed by a single contractor responsible for both software production and system integration. Costs could be assessed through augmentation to internal business accounting practices (e.g., budgeting, staffing workloads, time-sheet reports, project schedules, fixed fees, etc.). But a move to OA ecosystems means that multiple producers can participate, and OA schemes accommodate switching among providers, while a system is being integrated, deployed, or evolved in the field. This in turn coincides with new ways and means to electronically distribute software updates, components, or applications, as well as new ways to charge for software. For example, OSS components may be acquired and distributed at "no cost," but their integration and evolution charged as service subscription, or as time-effort billings.

We have already seen other alternatives for costing, paying for, or charging for software that include: franchising; free component, paid service/support fees; enterprise licensing; metered usage; advertising supported; subscription; federated reciprocity for shared development; collaborative buying; donation; sponsorship; government-provided free/open source software (e.g., Government OSS – *GOSS*); and others. So how are customer organizations, especially in the Defense Community where software cost estimation practices are routine, suppose to estimate the development or sustaining costs of the software components or integrated systems they acquire and evolve? How are software costs to be estimated when an OA system allows for producers whose components come with different costing/ billing schemes? This is an open problem for SE research in industry practice.

The last piece of the puzzle we are studying is the envisioned transition with the Defense Community to C2 system being composed by customer organizations, and possibly extended by end-users deployed in the field. This is the concept that surrounds the transition to discovering and acquiring software components, apps, or widgets in Defense Community app stores [6]. These app stores are modeled after those popularized for use in distributing and acquiring software apps for Web-based or mobile devices, like those operated by Apple, Google, Microsoft, and others. How the availability of such Defense Community app stores will transform the way C2 systems are produced and maintained, or even if they will be produced by legacy Defense industry contractors remains to be seen. Said differently, how app stores transform OA software ecosystem networks, business models, or cybersecurity practices is an emerging challenge for SE research.

DISCUSSION and CONCLUSIONS

In this paper, we have focused attention on software engineering research challenges that are emerging in the Defense Community. Much of the earlier research and advances in SE emerged from challenges in this same community 40-50 years ago. Most contemporary SE research has moved away from this community. However, as we sought to describe in this paper, this community is again surfacing and facing a growing myriad of issues and challenges that can directly benefit from targeted advances in SE research and practice.

We identified and examined six target areas for SE research that now plague the Defense Community, and perhaps other industries, along with other government agencies as well. All of these SE research areas are readily approachable, and research results are likely to have significant practical value, both within the Defense Community and beyond.

These issue areas were investigated and addressed in the domain of command and control systems (C2). We believe all these issues are tractable, yet dense and sufficient for both deep, sustained research study and also applied research in search of near-term to mid-term practical results.

In related work [15], we call for specific R&D investments into the development of open source, domain-specific languages for specifying open architecture representations (or ADLs, architectural description languages) that are formalizable and computational, as well as supporting annotations for software license obligations and rights. While ADLs have been explored in the SE research community, the challenges of how software architectures mediate software component licenses and cybersecurity requirements is an open issue, with practical consequence. Similar, ADL annotations that incorporate IP and cybersecurity requirements, along with costs or cost models in line with new software business models, is an open problem area. We have also called for R&D investment in new SE tools or support environments who purpose is to provide automated analysis and support of OA systems IP and cybersecurity obligations and rights, as new requirements for large-scale software acquisition, design, development, deployment, and evolution. Such environments are the automated tools that could be used to model, specify, and analyze dynamically configurable, component-based OA software systems expressed using the open source architectural representation schemes or ADLs noted here.We hope this paper serves to help throw light into these otherwise dark corners of SE research that can inform and add benefit to software development practices for C2 and enterprise business systems for use and evolution across the Defense Community.

ACKNOWLEDGMENTS

The research described in this report was supported by grants #N00244-15-1-0010 and #N00244-16-1-0004 from the Acquisition Research Program, at the Naval Postgraduate School, Monterey, CA. No endorsement, review, or approval implied. The concepts, topics, and materials presented are the sole responsibility of the authors.

REFERENCES

1. Alspaugh, T.A, Asuncion, H.A., and Scacchi, W. (2010). Software Licenses in Context: The Challenge of Heterogeneously Licensed Systems, *J. Assoc. Info. Systems*, 11(11), 730-755.
2. Alspaugh, T.A. and Scacchi, W. (2012). Security Licensing, *Proc. Fifth Intern. Workshop on Requirements Engineering and Law*, 25-28, September 2012.
3. Alspaugh, T.A. and Scacchi, W. (2013). Ongoing Software Development Without Classical Requirements,, *Proc. 21st. IEEE Intern. Conf. Requirements Engineering*, Rio de Janeiro, Brazil, 165-174.
4. Alspaugh, T.A., Scacchi, W., and Kawai. R. (2012). Software Licenses, Coverage, and Subsumption, *Proc. Fifth Intern. Workshop on Requirements Engineering and Law*, 17-24, September 2012.
5. Choi, S.C. and Scacchi, W. (1990). Extracting and Restructuring the Design of Large Systems, *IEEE Software*, 7(1), 66-71.
6. George, A., Morris, M., Galdorisi, G., Raney, C., Bowers, A., and Yetman, C. (2013). Mission composable C3 in DIL information environments using widgets and app stores. In *Proc. 18th Intern. Command and Control Research and Technology Symposium*, Paper-036. Alexandria, VA.
7. Kazman, R. and Carriere, S.J. (1998). Playing Detective: Reconstructing Software Architecture from Available Evidence. *J. of Automated Software Eng*., 6(2), 107-138.
8. Kendall, F. (2015). Implementation Directive for Better Buying Power 3.0, 9 April 2015. Also see Defense Acquisition University, *Better Buying Power*, http://bbp.dau.mil/
9. OSSI (2016). The Open Source Initiative, http://www.opensource.org/
10. Reed, H., Benito, P., Collens, J., and Stein, F. (2012). Supporting Agile C2 with an agile and adaptive IT ecosystem In *Proc.17th International Command and Control Research and Technology Symposium* (ICCRTS), Paper-044. Fairfax, VA.
11. Reed, H., Nankervis, J., Cochran, J., Parekh, R., Stein. F., *et al*. (2014). Agile and adaptive ecosystem: results, outlook and recommendations. In *Proc. 19th International Command and Control Research and Technology Symposium* (ICCRTS), Paper-011. Fairfax, VA.
12. Scacchi, W. and Alspaugh, T.A. (2012). Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems, *J. Systems and Software*, 85(7), 1479-1494, July 2012.
13. Scacchi, W. and Alspaugh, T.A. (2013b). Processes in Securing Open Architecture Software Systems, *Proc. 2013 Intern. Conf. Software and System Processes*, 126-135, May 2013, San Francisco, CA.
14. Scacchi, W. and Alspaugh, T.A. (2013c). Advances in the Acquisition of Secure Systems Based on Open Architectures, in *J. Cybersecurity & Information Systems*, 1(2), 2-16.

15. Scacchi, W. and Alspaugh, T.A. (2015). Achieving Better Buying Power Through Acquisition of Open Architecture Software Systems for Web-Based and Mobile Devices, *Proc. 12th Annual Acquisition Research Symposium*, Monterey, CA, May 2015.

BIOGRAPHIES

Walt Scacchi—is 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 200 research papers, and has directed more than 65 externally funded research projects. In 2011, he served as co-chair for the 33rd International Conference on Software Engineering—Practice Track, and in 2012, he served as general co-chair of the 8th IFIP International Conference on Open Source Systems.

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