Processes in Securing 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, thomas.alspaugh@acm.org

ABSTRACT

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

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Management, Security

Keywords

Open architecture, configuration, process modeling, process integration, continuous software development.

ICSSP '13, May 18-19, 2013, San Francisco, USA

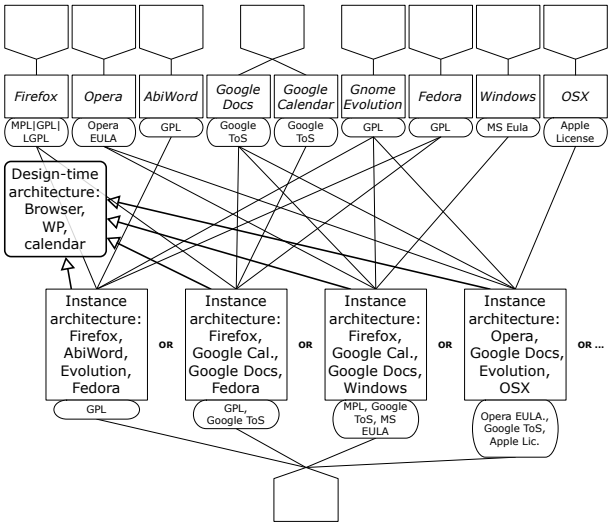

Figure 1: A software ecosystem of software components that can be configured into a product line indicating four functionally similar OA systems.

1. OVERVIEW

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

In this paper, we employ a case study focusing on an OA software system whose security must be sustained throughout its ongoing development and evolution. We limit our focus to software processes surrounding the architectural de-

sign, continuous integration, release deployment, and evolution found in the OA system case study. To be clear, these processes focus on activities that construct and update configurations of software components, and are not the processes for developing the components themselves. The components involved in such OA systems have their own development life cycle, often within development projects that are independent or at arm's length from the effort to develop and evolve an OA system composed from such components.

In our case study, we examine a simple OA enterprise computing system that configures a Web browser (Firefox, Opera, etc.), word processor (AbiWord, Google Docs, etc.), email and calendar component (Gnome Evolution, Gmail, etc.), and operating system (RedHat Linux, RedHat Fedora with SELinux, Microsoft Windows, Apple OSX, SEAndroid, etc.) in conjunction with file, mail, and Web servers (which may be on distributed network servers), in a loosely coupled manner. However, even this simple OA system that we study draws on an ecosystem of diverse software component providers, whose software products can be configured into alternative, functionally similar system configurations that conform to an OA software product family, as indicated in Figure 1. Such a OA system is also a core of more complex, mission-critical command and control systems [10, 31]. Additionally, such a system can also be built and deployed for use on a mobile computing platform like a tablet or smartphone. Finally, our OA system can be encapsulated within security capability and enforcement mechanisms (e.g., SELinux capabilities, virtual machine hypervisors) in order to secure the OA system [5, 32, 35, 38].

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

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

2. RELATED RESEARCH AND DEVELOPMENT EFFORTS

We choose to focus on the processes from architectural design, continuous integration, and release deployment to software evolution for OA systems. Such systems incorporate both CSS and OSS components. In particular, our interest is to examine how these processes enable or constrain how to produce a secure OA system. In particular, we recognized that processes for software architecture design and software evolution [22] have received prior attention in the software process community, but continuous integration and release deployment have received much less attention. Similarly, relatively little is known about how design processes enable and constrain continuous integration and delivery, nor how they in turn facilitate or constrain software evolution. Such an undertaking needs to go beyond prior efforts to specify and identify issues that may arise in processes for the development of component-based software systems [4, 27]. Earlier process studies like these do not address, for example, how new development technologies such as continuous integration systems mediate development processes for component-based systems. They also do not identify continuous integration, or software release delivery and installation, as salient development processes for component-based software systems. This may be so as continuous integration and release management are relatively new software development processes, and such processes seem to be visibly practiced in large OSS development projects. Finally, these earlier studies offer little insight as to how functional or non-functional requirements for securing an OA system mediate its software development and evolution processes. But we do know some things about these processes from related efforts, especially for continuous integration.

Continuous integration (CI) systems support automated processes for building, testing, and packaging a software system for release [7, 8, 36]. Without a CI system, developers must build, test, and integrate their software (component) products using hand-crafted scripts, and it is common for such scripts to have to rely on idiosyncratic dependencies on tool chains and libraries versions for each deployment platform targeted (e.g., [15]). In contrast, CI systems incorporate the capabilities of software build systems [33] that may invoke sequential, distributed, or parallel builds across multiple build servers (cf. [34]) to produce singular builds (e.g., "nightly builds"), continuously updated agile development builds [8], or diverse, functionally equivalent executable variants [17]. The build systems access and update software code (version control) repositories via process automation scripts. CI sub-processes take as input directories/folders of source code files and produce software component executables. The executables may also be organized as a structured collection (an information architecture) of binary files, static data value and parameter setting files packaged in interlinked directories, constituting releases for deployment. Continuous delivery (CD) further extends CI to support automated release management and the creation of automated deployment tools such as "installation wizards" to be used by system administrators or end-users [14]. For the remainder of our paper, we use the abbreviations CI and CD to refer to these sets of automatable software development processes

As Fowler [8] observed about the need for continuous integration as an enabling mechanism for agile development, "the key is to automate absolutely everything and run the process so often that integration errors are found quickly. As a result everyone is more prepared to change things when they need to, because they know that *if they do cause an integration error, it's easy to find and fix*" (emphasis added). CI processes can therefore be viewed with the assumption that errors resulting from process automation are normal, expected, and not necessarily easily anticipated. But why do these errors occur at all, and why do we need to run

the process often in order to identify and resolve integration problems? We need to make closer, systematic observations to determine why or how these errors occur, so that we can advance our process engineering knowledge, as well as to enable practical process improvement. A case study can serve as a starting point for this, and this is our strategy.

Automated CI systems comprise composed environments of software tools, or sets of loosely coupled tools together by automated process invocation scripts that guide and constrain their use. Often these tools are independently developed and evolved. For example, a CI system like Hudson [13] includes source code build tools like Ant or Maven, an issue tracking (or bug reporting) tool like Bugzilla [19] or Jira, and a software revision control browser and search engine like FishEye or ViewVC for viewing the contents of software revision control code repositories like CVS or Subversion. All of these tools happen to be OSS associated with active OSS development projects, so these tools are subject to ongoing development and evolution that improve their capabilities and add/remove functionality. Other CI systems may use different tools or locally developed capabilities in place of external OSS tools such as these. Consequently, this implies the process steps enacted by a CI system will vary (and evolve) depending on the choice of CI system, and on the external tools or locally embedded software functionality that particular CI system uses. Whether such CI process steps are equivalent, similar, or incongruent across CI systems thus remains an open issue. But it is an issue that must be resolved when transitioning from one CI system, or CI system version, to another. However, current CI systems do not appear to address this, nor do they identify it as a concern in their recommended best practices (cf. [13, 34]). Similarly, when we add the need to address the CI and CD of secure OA systems, we quickly finds gaps in the best practices that point to shortfalls either on the CI/CD process support side, the security capability side [35], or their interdependencies.

Automated CI systems are continuously being improved or supplanted [18, 21] and different CI systems offer different features, functional capabilities, and depend on different software tools [34]. The same can be said for CD/release deployment systems, especially with regard to ongoing advances and refinement of software packagers, file distribution and mirror (copy server) synchronization, installers, and uninstallers [14]. So from a software process specification or modeling viewpoint, there are many distinct CI process instance types, and no single abstract CI or release deployment process prescription to follow and tailor to local development organization needs. CI and CD process enactment must therefore rely on manual best practices in addition to tool-based automation, and these practices are specific to each CI system and the tools therein [13]. CI and release management system-based process automation thus is both *ad hoc* and idiosyncratic, rather than easily standardized or generalized, yet is a widespread software engineering process and practice used to produce thousands of software components (e.g., smartphone or tablet apps).

Software delivery and deployment suffer similar kinds of process automation pathologies (e.g., [16]), to the extent that a key advantage of automation is now thought to be finding or process enactment errors, mistakes, or other articulation problems [23] by running the enactment more quickly. Software deployment errors, such as releasing and installing a premature system release candidate into production operations can have devastating technical or economic consequences, as was demonstrated by the experience of Knight Capital in Summer 2012 [6]. How to provide automated tools and practical techniques that provide (more) robust acceptance/compliance checking prior to a new system version being installed prior to going live in operation, seems to be an underspecified process enactment problem. Adding robust diversity mechanisms and capabilities for dramatically improving OA system security [11, 17, 30] remains an open question for further study. Once again, a case study can serve as a starting point for examining such issues and concerns, and this is our strategy.

We see that part of the process challenge is how to understand and specify software processes that must interface with emerging CI and CD systems. These CI systems entail different kinds with different build, package, and release deployment process automation capabilities, or that produce integrated systems that operate on different platforms [34]. To us, this raises concerns for process specification—determining what aspects of a software process are pertinent for modeling and simulation, as well as contributory to improving process effectiveness [26], and process integration—integrating modeled process specifications with diverse automated process enactment mechanisms [24]. It also raises issues for integration across multiple process representations that are supported by independently developed, heterogeneous process enactment mechanisms [9].

3. CASE STUDY: A SECURE OA ENTERPRISE SYSTEM

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

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

3.1 Architectural Design Process

The process for designing the configuration of an OA system at the component level is our focus here. We start by noting that we assume no pre-existing process model or standard for such a process, nor do we propose to provide such a prescriptive process. As a review of the architectures of dozens of OSS systems [3] makes clear, there is no common prescriptive process, preferred set of tools, nor is there notational scheme for the architectural design of open software systems. Instead, we describe aspects of a design process we developed, practiced, and adapted that is supported in part with automated design tools. One of our goals with this process was to help identify situations, and practical non-functional requirements, that arise with an OA design process that constrains, and is constrained by, the other three downstream software processes in our study.

We have used an OA tailored version of the UCI ArchStudio4 architecture design system (oAS4) as a locally developed plug-in to the Eclipse IDE to realize a partially automated system for architectural design activities [1, 2]. oAS4 allows us to visually model the architectural configuration of software components, component interfaces, and component connectors as OA system elements. oAS4 also produces output in an architectural description language (ADL) as a persistent artifact for external analysis, or for potential integration with CI systems with further processing (e.g., binding component classes to their build-time instances). We further focus our architectural design activities to produce an abstract system architecture that serves to denote a product line model of a family of alternative system configurations composed from functionally similar components or component versions [29]. oAS4 can thus support our experimental studies in OA system design and design evolution across families of alternative system configurations (cf. an earlier approach to such problems at [25]).

We annotate our OA system designs within oAS4 using formal constraint expressions on components interfaces, such as intellectual property (IP) license obligations and rights [1, 2]. Security policy constraints for components, configured sub-systems, or an overall system are expressed and analyzed in a similar manner [30]. The ability to model and automatically analyze such obligations and rights is needed at build-time and release deployment-time. Automated analysis mechanisms then allow us to determine whether the specified component interconnections entail matches or conflicts in component-component license alignments [1, 2]. However, we have also observed that design-time actions must accommodate build-time and deployment-time element bindings, as well as accommodate the evolution of licenses, policies, and system element versions [29]. For example, when conflicts are found between the licenses of interconnected build-time component selections, we can then reconfigure our OA system design to eliminate the conflicts, to constrain the selection of components at build-time (within CI) to those whose licenses will match or not conflict, or to wrap/shim a component with an abstraction layer that does not transfer IP license obligations.

Design of OA systems also raises issues for how to how best to secure the designed system architecture [35]. Among the recommended practices for designing secure system architectures are to provide capability-based user/developer access control that effectively limits access to input and output data, internal program code representations (e.g., memory address and system name spaces), persistent data storage, and to exposed I/O transaction processing interfaces. One increasingly common approach is to provide encapsulation mechanisms like virtual machines for software components or (sub-)system configurations, along with encrypted inter-component data/control flow connectors (e.g., HTTPS/SSL data communication protocols). Of these, passively secure connectors for networked components are widely available, while dynamically secured connectors are a recent advance [11]. In our case, we choose to incorporate virtual machines to encapsulate our OA system, and we ignore alternative security protection schemes for simplicity. However, we recognized that even a seemingly simple decision like this still requires analyzing trade-offs about whether to encapsulate the entire system as a single virtual machine (relatively easy to address during deployment, though requiring deployment and installation of virtual machine software (e.g., [38]) on the target deployment computers) or to encapsulate each different component within its own virtual machine that would then be interconnected using secure connectors (more challenging to address for deployment, but offering a more resilient OA system security [30]. We decided to design something in-between these two extremes, by taking into account where different components might be hosted within a networked, multi-server platform environment. What our OA system design process produced is an abstract architectural configuration of component types (each attributed with IP license constraints—not shown but described elsewhere [1, 2, 30]), a minimal component interconnection scheme, and what we call a hybrid virtual machine confinement scheme, as shown in Figure 2.

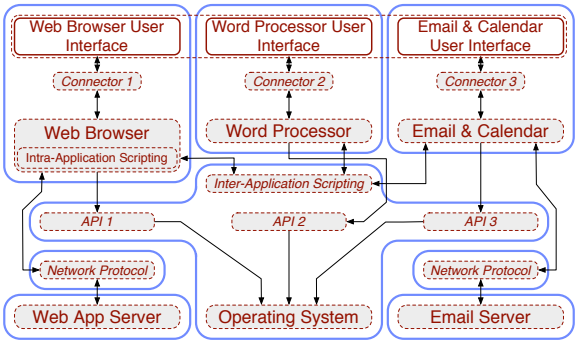

Figure 2: Design configuration of a secure OA enterprise system, shown with security encapsulation layout.

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

3.2 Continuous Integration Process

In our study, one of the first activities in moving from architectural design to continuous integration is to identify specific software component versions that can be instantiated within the current architectural configuration (Fig-

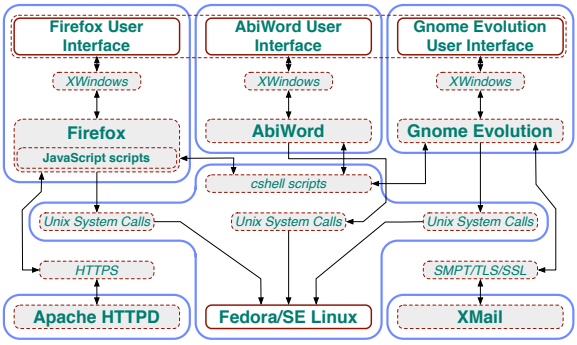

Figure 3: A build-time configuration of a secure OA enterprise computing system.

ure 2). While at first it might seem that this is a simple task, we have found that component and version selection are subject to the obligations and rights stipulated with a component's associated IP license [1]. For example, common architectural design languages do not specify annotations for IP licenses, so as noted above, we extended our ADL within the oAS4 with IP obligation and right constraints [1, 2]. This meant we could now analyze whether or how IP obligations and rights for each component-component interconnection match, conflict, or propagate. For example, reciprocal licenses like GPL can propagate their IP regime by design, though some enterprises seek to avoid this. By conceptually filling in selected component licenses, we can tell, prior to integration, whether the resulting release candidate may suffer from licensing problems or not. When conflicts or mis-matches are discovered, again prior to further build-time process actions, alternative components with the similar functional capabilities and interfaces but different licenses may be substituted. Alternatively, the architectural configuration can be modified, for example, wrapping a component in a way that mitigates license conflicts (e.g., replacing a direct API-API interconnection which propagates license restrictions with an networked data communications link, as few licenses propagate IP across network connections).

What we end up with from our build sub-process is a concrete OA system configuration with a specific selection of software components specified using oAS4, whose output is intended for a manual build system or for entry into an automated CI system. A concrete configuration is seen in Figure 3. So our build sub-process can now instantiate components into a reusable OA software product line design, as we can determine families of component version instances that can be substituted within the OA system. For example, the Firefox Web browser may be replaced by Google Chrome in this configuration, because both are under permissive OSS licenses. However, a license match/conflict assessment would be required before replacing Firefox with Microsoft Internet Explorer (IE) or Opera, each of which is under a proprietary license. But in the abstract and concrete architectural configuration we have, we could substitute a Linux-based Opera browser without issue, but not IE, unless we add a library wrapper such as Wine [37], in order to run IE on Fedora Linux.

So far, so good. But now we must consider how to transfer this component selection specification into the build system arises. An ideal solution might involve an automated hand-off. However, the specifics of such a hand-off will vary depending on the build system and the CI system we select. A more general solution would likely require (or benefit from) another abstraction layer for integration between the architectural design and build/CI process enactment mechanisms, which is an already recognized problem with a demonstrable solution (cf. [9]). We see that software process research may demonstrate solutions to messy process integration issues, but integration of process flows across tool-specific process enactment representations and automated mechanisms remains a lingering, practical problem that is not yet addressed by current CI or CD systems.

A similar problem arises when we consider how to secure the concrete OA system configuration. For example, we can choose to include secure data communication connectors (e.g. secure protocols like HTTPS and TLS/SSL) in our configuration, but such capabilities are not instantiated at build-time. Instead, they depend on mechanisms and data (e.g., certificates) that are accessed at run-time once an integrated system release candidate is available. An OA system, or OA system components, can also be secured using virtual machine hypervisors [38] that confine and isolate deployed system/component within a virtual machine run-time environment. In addition, it should be possible to specify operating system access control and type enforcement capabilities (e.g., using SELinux libraries on Fedora), but again, these are not available for use until there is a deployable integrated system release candidate. Thus, these forms of security are most likely invisible to current CI systems, and must be addressed through other means.

3.3 Release Deployment Process

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

However, what we build and what we release may not be the same, though they need to be functionally equivalent. For example, when we select one or more CSS components (an already compiled and integrated executable binary image) with a common restrictive IP license (i.e, one that prohibits copying or redistribution) for inclusion in our build-time architectural configuration, during the build process, we must link it as an executable binary for inclusion in a release candidate for deployment (or deployment testing) (cf. [19]) on a local computer. Such inclusion is a prerequisite for overall integrated system testing processes required by CI. However, we cannot distribute such a release candidate to others, as it is common for CSS to not allow duplication or distribution of licensed copies of software binaries. Instead, we need to specify and configure a deployment-platform specific automated software installation mechanism (e.g., installation wizard) that needs to search for and find a local licensed copy of the CSS executable binary, and link it to the result of the build sub-process that provides a run-time linkage mechanism in expectation. A similar effort is needed

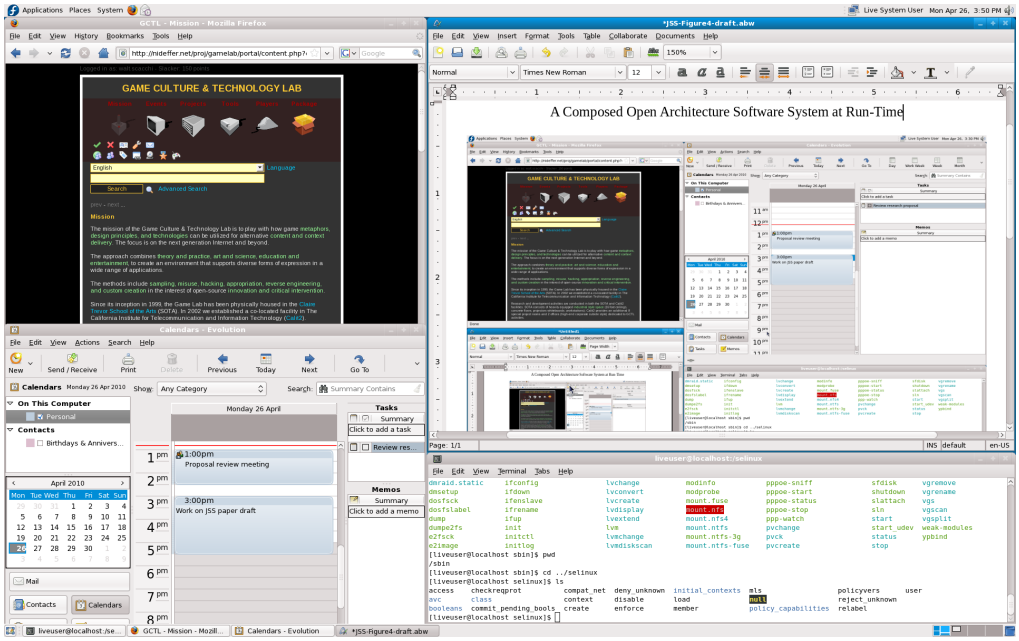

Figure 4: A screenshot view of a deployed release configuration of our OA enterprise computing system.

to enable user acceptance testing or certification testing on their local platform. These release deployment process steps can be accomplished with some effort, but this effort could also be anticipated at design-time or build-time, when developers make their selection for which component instances to include in the system build.

Automated software installation is an increasingly common expectation. Software installers run automated process enactment scripts crafted by developers. Once again, the process being enacted is not explicitly specified, nor is it separate from the internally coded software utility's action invocation scripts. This means that it is not surprising to discover errors that arise during installation but are not easily anticipated without extensive prior experience in working with the installer on known target platforms. For example, an informal aid from IBM for guiding system administrators who enact software installation processes [16] notes installation problems like: (a) insufficient free space on disk storage prior to or during software executable installation; (b) software installations across a network that are "hung" or stuck due to lack of robust installation protocols that can time-out (abort) and/or re-initiate then re-validate process script commands already invoked; (c) installations that fail due to underspecified all/nothing installation transactions (cf. [12]) that do not completely update the information architecture of a multi-part software configuration (e.g., program registry update and reversible roll-back to prior registry; and/or setup of user configuration files); (d) failure to include a software uninstaller (or uninstallation process scripts) that allows conditional roll-back to previously installed software versions to be retrieved and activated; or (e) file/directory name collisions that arise at build-time versus deployment-time.

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

3.4 Evolution Process

An OA system can evolve by a number of distinct mechanisms or process enactment pathways, some of which are common to all systems, but others of which arise only in OA systems or where components in a single system are heterogeneously licensed [2]. Figure 5 provides a summary of some of the various paths, further explained below.

Component version evolution— One or more components can evolve, altering the overall system's characteristics. An example is upgrading the Firefox Web browser from version 17.0 to 17.1. Such minor versions changes generally have no effect on system architecture. However, many large enterprises choose to sustain their software systems by relying on "long-term support" (LTS) versions of software components, rather than automatically updating to each release from software component producers. Instead, LTS components are replaced with new versions only over long time frames, where the new LTS version for installation may skip many intervening release versions. Such enterprises rely on local patches and workarounds between the LTS versions, under the belief that such an approach provides increased system stability and allows more comprehensive regression

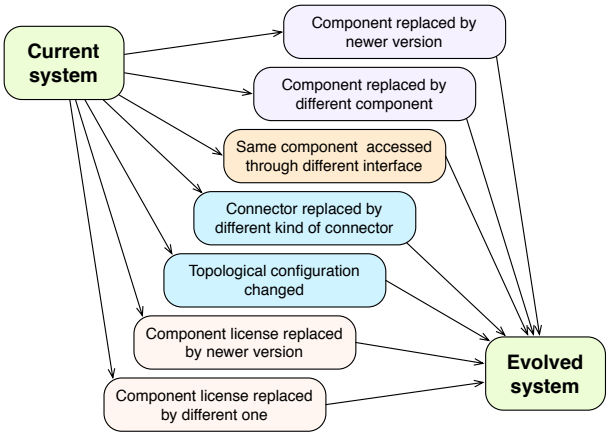

Figure 5: A variety of paths and activities accounting for the evolution of OA systems [29].

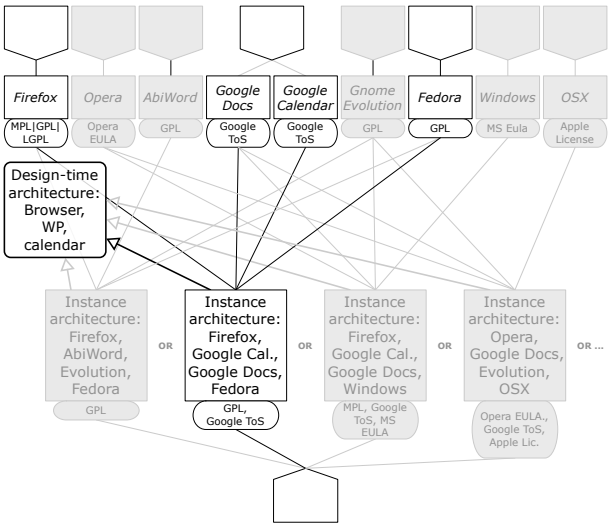

Figure 6: An alternative OA system configuration resulting from replacement of selected components shown in Figure 4 during system evolution [29].

testing prior to deployment. But in these days of relentless attacks on system security, using LTS components entails locally sustaining system component or configuration versions with known vulnerabilities, often without code repositories that match those employed for CI. The vulnerabilities must then be defended using separate, orthogonal system security mechanisms, such as virtual machines or hypervisors from VMWare or Xen [38]. Once again, we can do better than this through the use of explicit process specifications that model and provide process integration support across CI and CD systems, along with code repositories.

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

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

Component license evolution—The license under which a component is available may change, as for example when the Mozilla core components changed from dual licensing to the tri-license (MPL, GPL, LGPL). Similarly, when Oracle Corporation took ownership of the Hudson CI system [21], the changes in intellectual property ownership and branding precipitated a major code fork, and instigated parallel independent projects for sustaining development of this OSS CI system [13, 18]. Such evolutionary changes, which are common to OSS components, may require reconfiguring an OA system to migrate to a new (re-licensed) component version, or to an alternative system configuration [29].

In response to different desired rights or acceptable obligations— The OA system's integrator or consumers may desire additional license rights (for example the right to sublicense in addition to the right to distribute), or no longer desire specific rights; or the set of license obligations they find acceptable may change. In either case the OA system evolves in response, whether by changing components, evolving the architecture, or other means, to provide the desired rights within the scope of the acceptable obligations.

Rapid dynamic system reconfiguration— More ad-

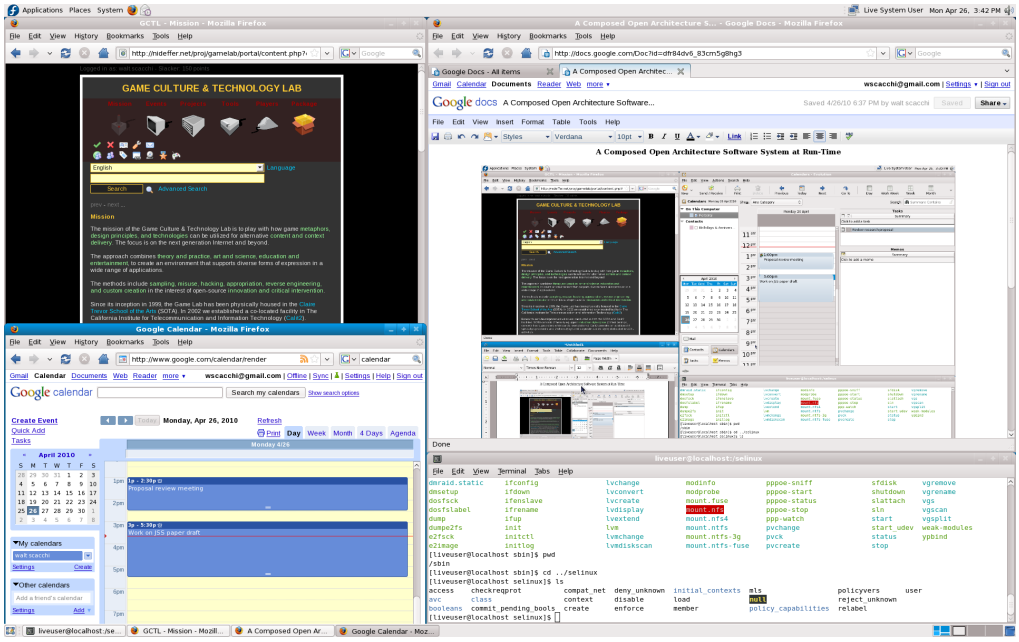

Figure 7: A screenshot view of a deployed release configuration of the alternative OA system configuration resulting from system evolution [29].

vanced evolution scenarios entail support for building and releasing of multi-variant system deployment configurations that substitute functionally equivalent software component compilations that produce multiple, diverse executable binary images, each of which may execute in its own processor core, in a multi-threaded, multi-core processor [17]. Pursuing this new path requires a new compilation and build system regime, that in turn anticipates a new generation of CI and CD systems as future research subjects.

As should be clear, our purpose is not to provide a prescriptive model of the OA system evolution process, but instead to illuminate how different OA system evolution paths and activities point to issues in process specification, process integration, and the integration of different process enactment representations and mechanisms that must span/link manual-to-automated process hand-offs.

4. OVERALL OA DEVELOPMENT AND EVOLUTION PROCESS ISSUES

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

First, we find that a central goal of process automation with widely available software integration and release deployment tools is to find enactment errors and articulation problems more quickly, rather than to provide prescriptive process guidance. Such process enactment details cannot be easily anticipated in general, so process specification and enactment must rely on trial and error, as well as process discovery [20] to surface where additional/new process knowledge is to be found. Consequently, it is not surprising to observe the rise of a new class of software developer role, as "buildmeisters"—developers who specialize in addressing the intricacies, quirks, and problems that arise during software integration processes, since such processes remain *ad hoc*, undefined, and difficult to model or improve.

Second, current continuous software development systems embody process specifications that are opaque, lack generality, and rely on the processing capabilities of specific incorporated tools to structure process enactment actions, decisions, and outcomes. Different CI systems embody different versions or variants of software build, test, and package processes. This implies that merely having a "defined" process model for processes like continuous integration and release deployment means that such a model will either be insufficiently detailed to provide anything beyond introductory level guidance, or more completely detailed but idiosyncratic because it is bound to specific process automation tools. This in turn makes the process specification problematic to adapt and evolve. There is a basic need for richer process models that represent both the idiosyncratic details of process automation tools for continuous integration and release management, and the generalized abstractions of such processes that can be reused for process (design) guidance and tailoring in specific software development organization settings (cf. [26]).

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

Fourth, automated process enactment systems are themselves subject to continuous improvement and evolution. This means the processes being supported are potentially evolving. However, if their process specifi

cation or model is tacit, or is encoded in implementation details, then the process may be opaque to all except the tool's developers. Thus, trying to specify, model, or simulate software processes that employ automated enactment systems, requires the ability to address processes that are co-

evolving: i.e., how tool evolution drives development process evolution, and how development process evolution precipitates tool evolution (cf. [28]). So choosing to only attend to one, misses observation or specification of activities that enable or constrain the other. Such a dilemma points to another challenge for new software process research.

Fifth, process guidance specification and enactment automation are easily conflated in continuous integration and release deployment systems. As a result, developers of OA systems rely on informal best practices to get continuously-integrated software products out the door. Separating the specification of such processes from their implementation within the automated system would be an important contribution to the advancement of such systems. Similarly, providing guidance for how to specify processes more abstractly than as low-level process execution script commands (cf. [15]), would also contribute to the advancement of automated continuous software development systems.

Sixth, the development and evolution of component-based OA systems is both an interesting and a challenging problem for the software process research community. Such systems are likely to follow continuous software processes—processes that are repeatedly enacted hundreds to thousands of times during the sustained life of the system. Such processes are thus appropriate for careful empirical study, simulation, and analysis. The need to address how to continuously secure OA systems further complicates the challenges for software process research. Process streamlining optimizations, opportunities, and guidelines are likely subjects for further research and practical application. Similarly, when the software processes for securing an OA system involve automated process enactment, it appears that compliance testing—checking whether an automated enactment produced a system configuration that is compliant with the system's security policy—will increase in importance. Such compliance is likely to be *ad hoc*, unless the security policy is formalized into a computational model [30] that can be cross-checked with the enactment results.

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

5. CONCLUSION

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

Managing and coordinating the development and evolution processes for producing secure open architecture software systems is challenging as we have shown in our case study. But as we have observed in our case study, widely available automated technologies for continuous integration and release deployment obscure or hide what these processes are. Further, we find that frequent errors and articulation problems in automated process enactment are expected, since process enactment details are *ad hoc* and idiosyncratic, while enactment processes are underspecified, not explicit, and encoded in an enactment system's implementation. However, automated process enactment systems may offer the potential to be extended to support (partially) automated process discovery and computational re-enactment (cf. [20]), rather than just traditional process modeling and simulation. Thus, software producers of contemporary component-based OA systems are working against their self interests, assuming their interests are to improve their productivity and software quality, while reducing avoidable rework and other software production cost drivers.

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

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

6. ACKNOWLEDGEMENTS

This research is supported by grant #N00244-12-1-0067 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #1256593 from the U.S. National Science Foundation. No review, approval, or endorsement implied.

7. REFERENCES

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

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

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

[4] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and

component lifecycle. In *Int. Conf. on Software Eng. Advances (ICSEA '06)*, pages 44–54, 2006.

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

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

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

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

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

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

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

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

[13] Hudson-ci. Hudson best practices. Eclipsepedia, Aug. 2011. http://wiki.eclipse.org/Hudson-ci/Hudson_Best_Practices. Accessed 2 Jan 2013.

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

[15] Hypertable. How to build Hypertable on various platforms, 2013. Accessed 1 Dec 2012. https://code.google.com/p/hypertable/wiki/HowToBuild.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[35] US-CERT. *Architecture and Design Considerations for Secure Software Development*. Software Assurance Pocket Guide Series. U.S. Dept. of Homeland Security, 2011. https://buildsecurityin.us-cert.gov/swa/downloads/Architecture_and_Design_Pocket_Guide_v1.3.pdf.

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

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

[38] Xen hypervisor project. http://xen.org/products/xenhyp.html.