VOLUME 1 NUMBER 2

Journal of Cyber Security & Information Systems

Secure Systems and Software Productivity

Advances in the Acquisition of Secure Systems Based on Open Architectures

By Walt Scacchi and Thomas A. Alspaugh

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. But 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", as well as 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.

Introduction

A substantial number of development organizations are adopting a strategy in which a software-intensive system is developed with an open architecture (OA) [20], 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 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 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 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 to accommodate. Strategic systems may need the greatest security possible, while 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 system 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. These capabilities thus 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 either (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 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 versions numbers like Firefox 3.6.16.144 (and possibly timestamps of their creation or local installation) arises since evolutionary version updates often include security patches that close known vulnerabilities or prevent known exploits. As indicated in the Related Research section below, 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.

As 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, as well as be automatically analyzed or tested in ways we have already established [4,5]. 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 find 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 software acquisition ecosystem producing and integrating secure software components or secure systems is portrayed in Figure 1.

In order to most effectively use an OA approach in developing and evolving a system, it is essential to consider this OA ecosystem. An OA system draws on components from proprietary vendors and open source projects. Its architecture is made possible by the existing general ecosystem of producers, from which the initial components are chosen. The choice of a specific OA begins a specialized software ecosystem involving components that meet (or can be shimmed to meet) the open interfaces used in the architecture. We do not claim this is the best or the only way to reuse components or produce secure OA systems, but it is an ever more widespread way. In this paper we build on previous work on heterogeneously-licensed systems [15, 22, 2] 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.

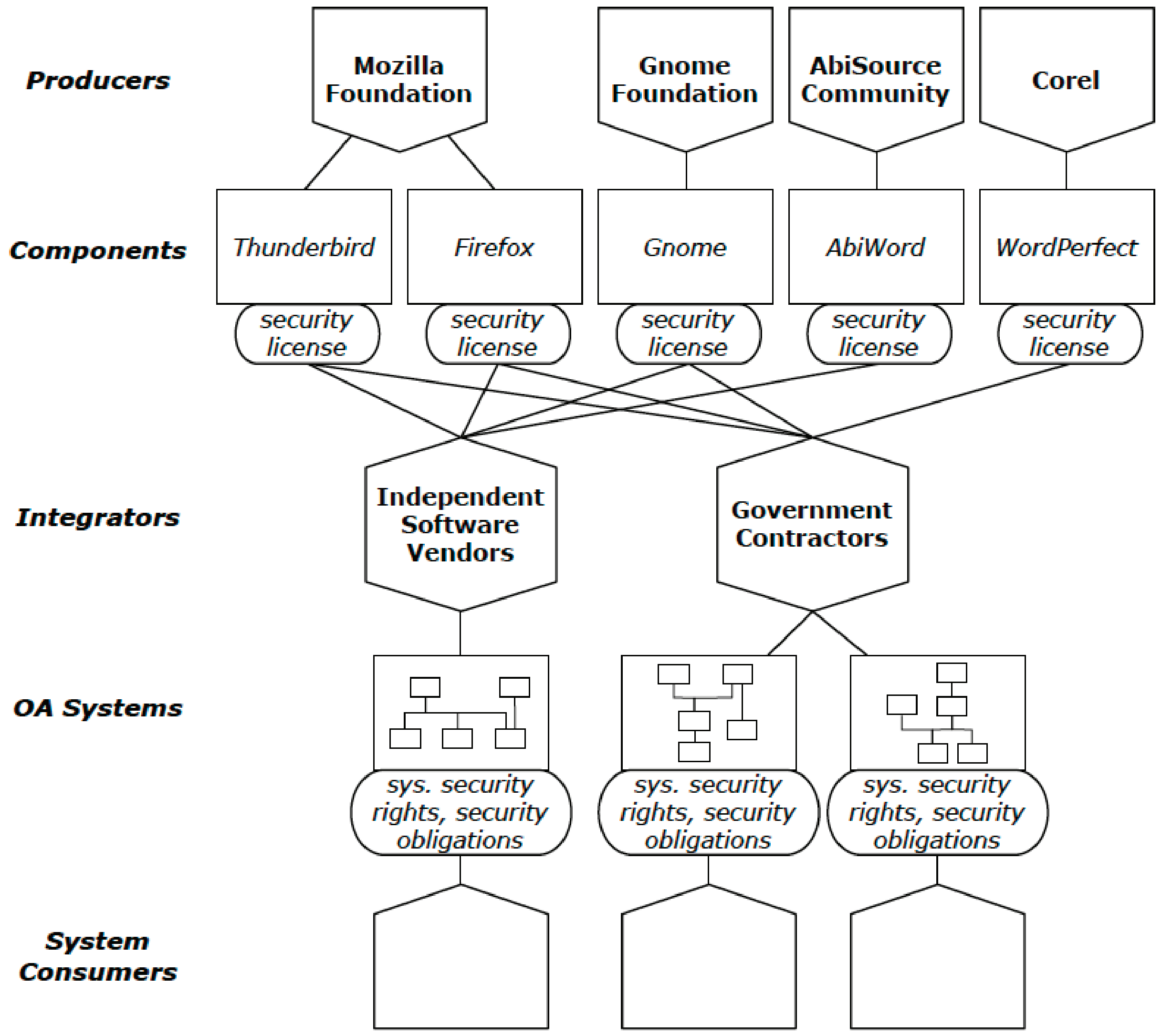


Figure 1: An example of a software ecosystem in which secure OA systems may be developed

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 products lines (Section 8), before stating our conclusions (Section 9).

2 R elated 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 by 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 [11] 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 (b) "sandboxing" (i.e., isolating) and holding up any evolutionary updates (the attacks) prior to their installation and run-time deployment. Furthermore, as the Stuxnet attack involved the use of corrupted certificates of trust from approved authorities as false credentials that allowed evolutionary system updates to go forward, it seems clear that additional preventions are needed that are external to, and prior to, their installation and run-time deployment. In our case, that means we need to specify and analyze software security requirements and evolutionary update capabilities at architectural design-time and system integration built-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 systems or system components as intellectual property (IP) or security requirements (i.e., capabilities) during their development and deployment. But rights and obligations are not limited to concerns or constraints applicable only to software as IP. Instead, they can be written in ways that stipulate non-functional requirements of different kinds. Consider, for example, that desired or necessary software system security properties can also be expressed as rights and obligations addressing system confidentiality, integrity, accountability, system availability, and assurance [8, 9]. 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 [23]. Using a semantic model to formally specify the rights and obligations required for a software system or component to be secure [8, 9, 23] 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 [3, 4, 5, 22], 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 [13] 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.

3 Secure Open Architecture Composition

Open architecture (OA) software is a customization technique introduced by Oreizy [20] 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 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 [22]. 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 incorporate secure OSS and proprietary components, and uses open APIs, will result in a secure OA system. But not all such architectures will produce a secure OA, since 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 [22, 1].

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed [6].

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, as for creating Rich Internet Applications using domain-specific languages such as XUL for the Firefox Web browser [12] or "mashups" [19]. 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 [21]. 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.

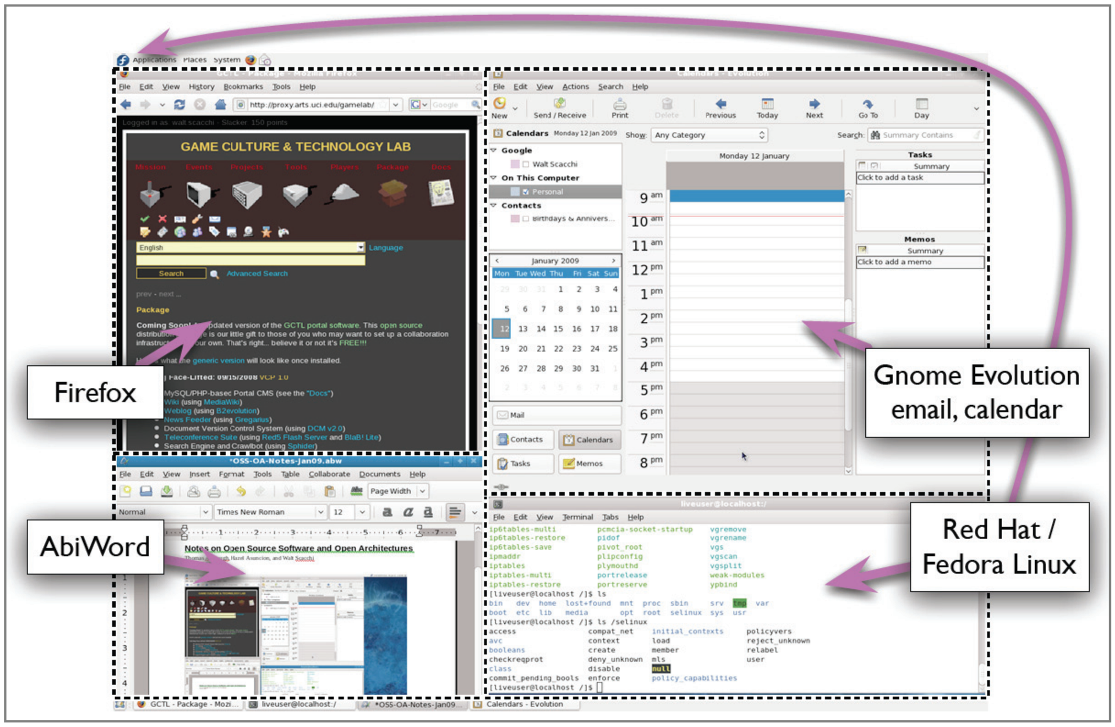
Application programming interfaces/APIs—Availability of externally visible and accessible APIs is the minimum requirement for an "open system" [18].

Software connectors—Software whose intended purpose is to provide a standard or reusable way of communication through common interfaces, e.g. High Level Architecture [17], 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 the kinds of software elements listed above. 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, 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. 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.


Firefox
Gnome Evolution email, calendar
AbiWord
Red Hat / Fedora Linux
GAME CULTURE & TECHNOLOGY LAB
liveuser@localhost:/

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

Figure 2: An example composite OA system potentially subject to different IP and security licenses

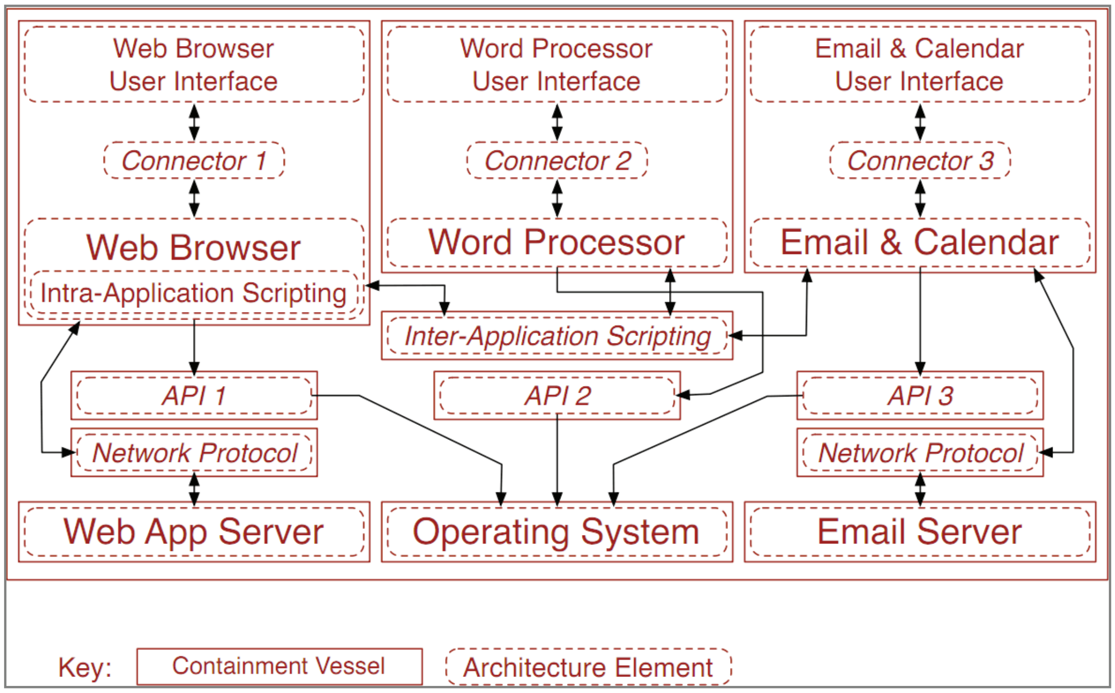


Figure 3: The design-time architecture of the system in Figure 2 that specifies a required security containment vessel scheme.

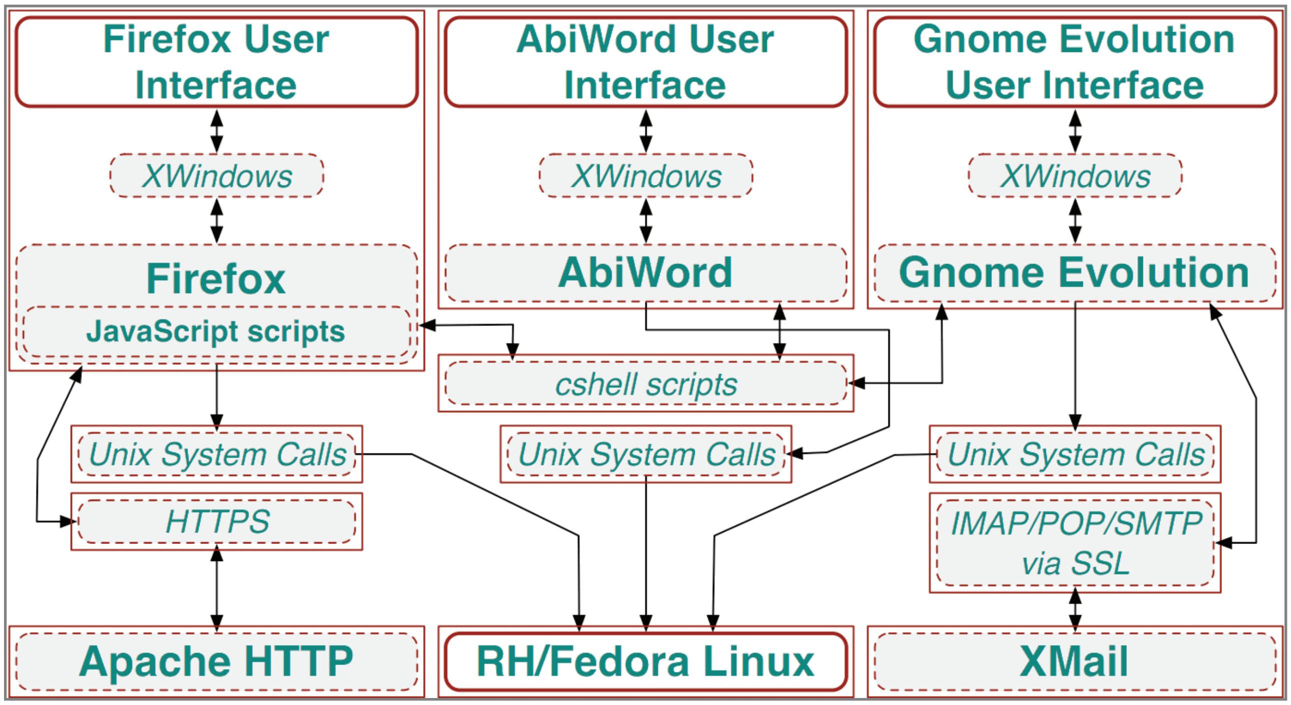


Figure 4: A secure build-time architecture describing the version running in Figure 2 with a specified security containment vessel scheme.

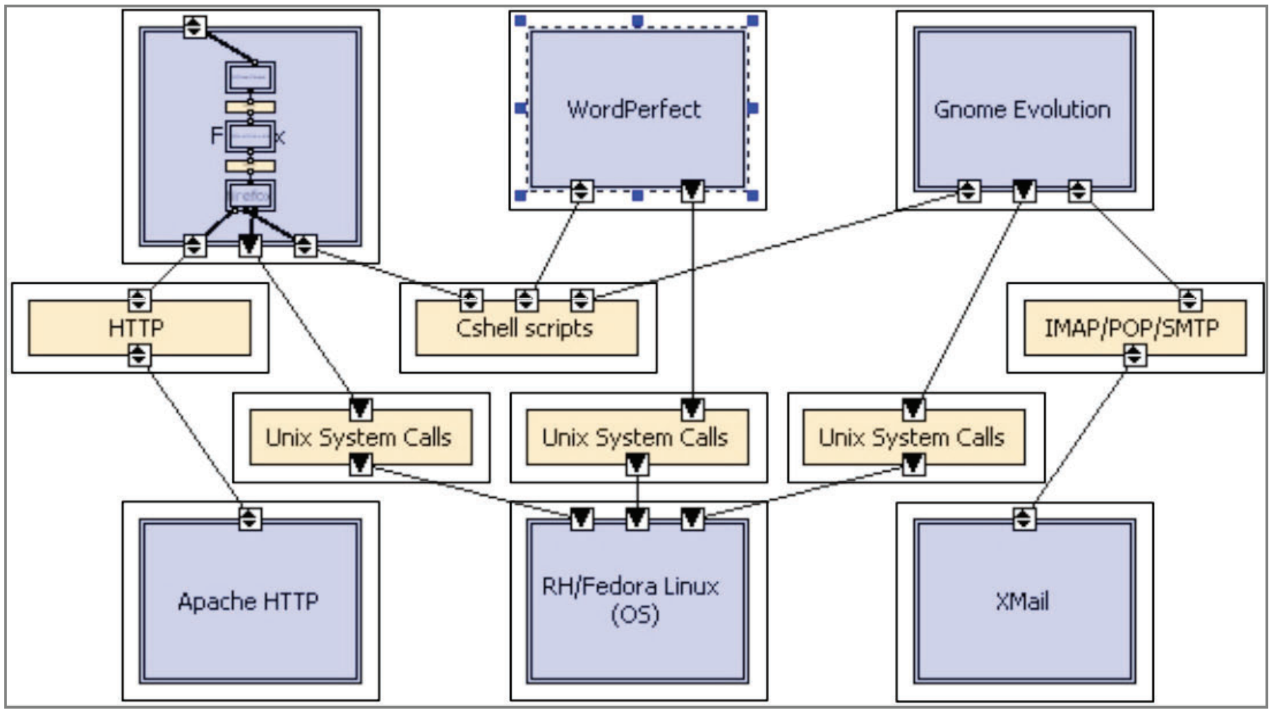


Figure 5: Instantiated build-time OA system with maximum security architecture of Figure 4 via individual security containment vessels for each system element.

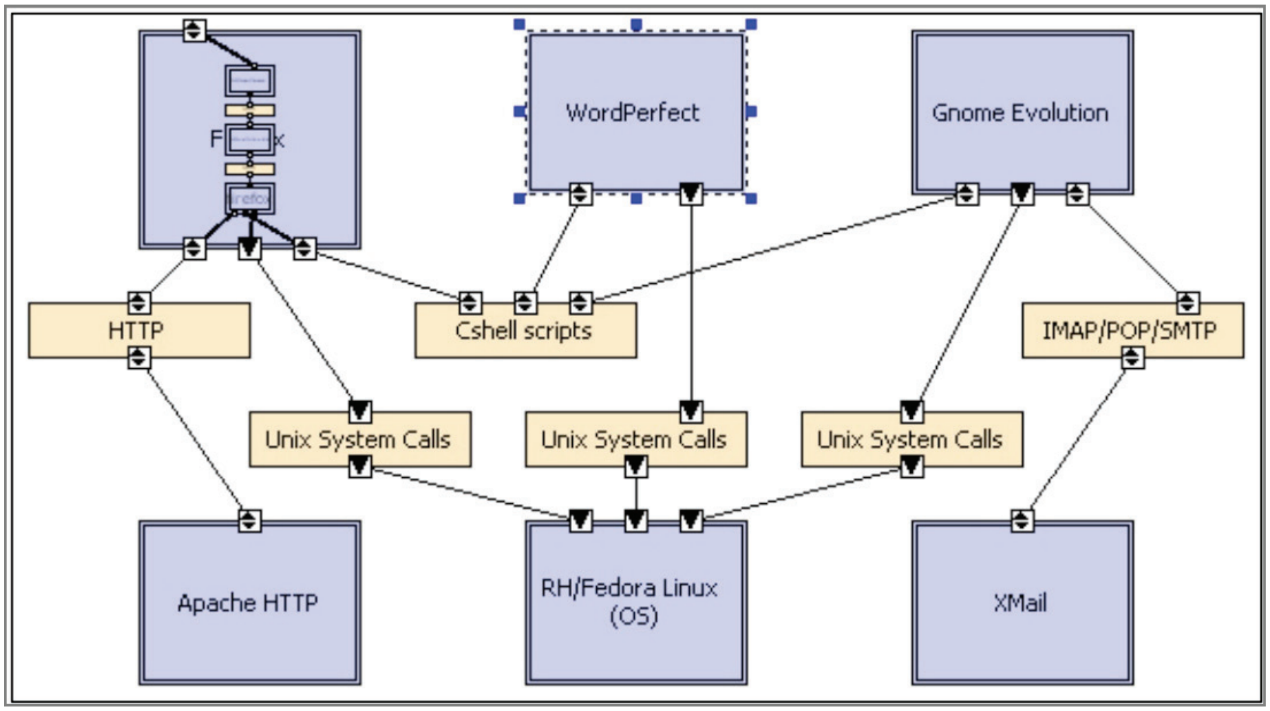


Figure 6: Instantiated build-time OA system with minimum security architecture of Figure 4 via a single overall security containment vessel for the complete system using a common software hypervisor, such as Xen, KVM, or VMware.

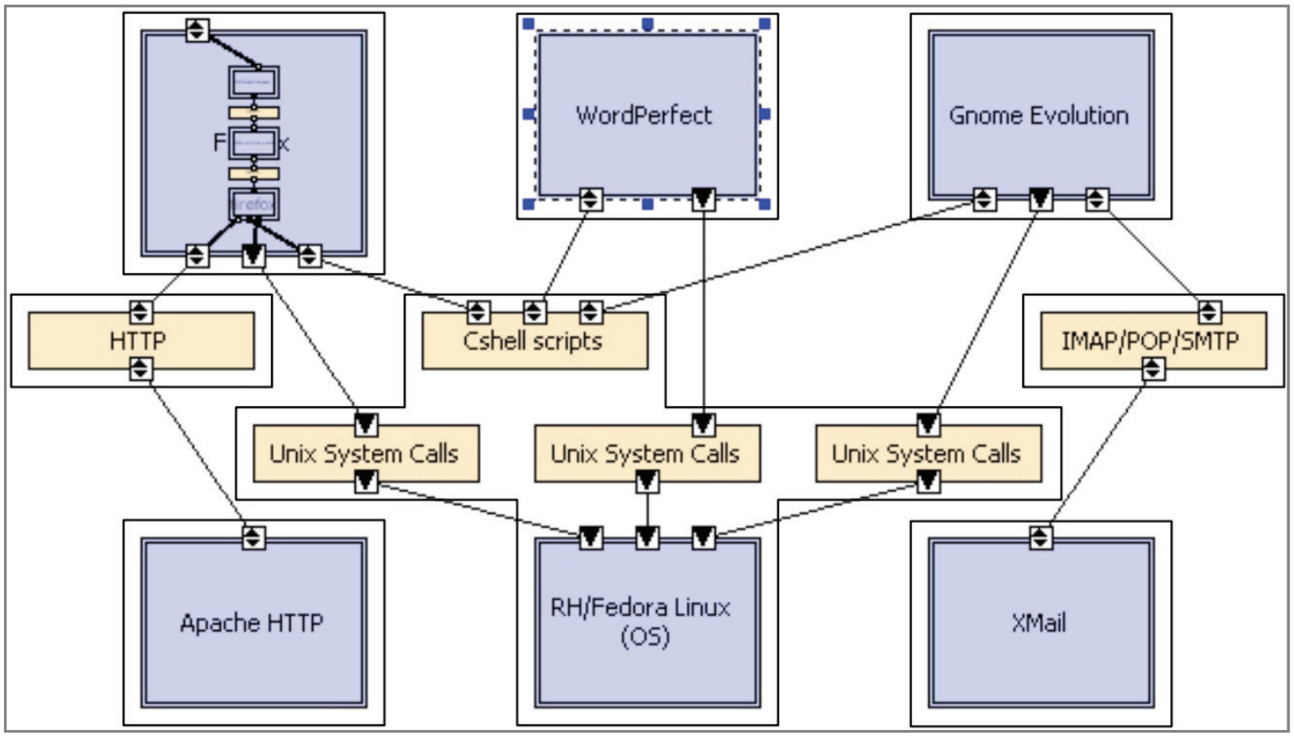


Figure 7: Instantiated build-time OA system with mixed security architecture of Figure 4 via security containment vessels for some groupings of system elements.

4 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.

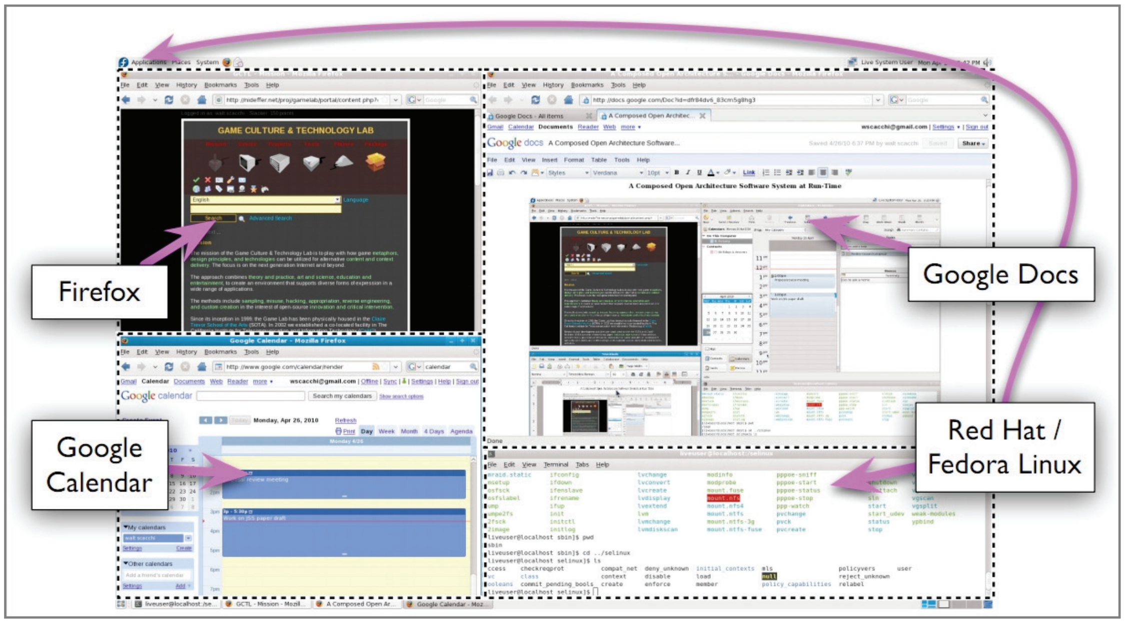
By component evolution— One or more components can evolve, altering the overall system's characteristics (for example, upgrading and replacing the Firefox Web browser from version 3.5 to 3.6 which may update existing software functionality while also patching recent security vulnerabilities).

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 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).

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 since the Google services (operating in a cloud environment) may not be easily accessed or penetrated by a security attack.

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.

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, 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.


Firefox
Google Docs
Google Calendar
Red Hat / Fedora Linux
GAME CULTURE & TECHNOLOGY LAB
Google docs
A Composed Open Architecture Software System at Run-Time
Google calendar
Monday, Apr 26, 2010
File Edit View Terminal Tabs Help

```
mraid.static   ifconfig     lvchange      modinfo          pppoe-sniff
msetup         ifdown       lvconvert     modprobe         pppoe-start
osfsck         ifenslave    lvcreate      mount.fuse       pppoe-status
osfslabel      ifrename     lvdisplay     mount.nfs        pppoe-stop         sln        vgscan
ump            ifup         lvextend      mount.nfs4       ppp-watch          start      vgsplit
umpe2fs        init         lvm           mount.ntfs       pvchange           start_udev weak-modules
2fsck          initctl      lvmchange     mount.ntfs-3g    pvck               status     ypbind
2image         initlog      lvmdiskscan   mount.ntfs-fuse  pvcreate           stop
liveuser@localhost sbin]$ pwd
sbin
liveuser@localhost sbin]$ cd ../selinux
liveuser@localhost selinux]$ ls
ccess    checkreqprot          compat_net  deny_unknown  initial_contexts  mls                   policyvers      user
vc       class                 context     disable       load              null                  reject_unknown
ooleans  commit_pending_bools  create      enforce       member            policy_capabilities   relabel
liveuser@localhost selinux]$
```

Figure 8: A second instantiation at run-time (Firefox, Google Docs and Calendar operating within different Firefox run-time sessions, Fedora) of the OA system in Figures 2, 3, and 4 as an evolutionary alternative system version, which in turn 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 create a tension through the ecosystem relationships with consumers and their releases of OA systems using those components, as integrators accommodate the choices of available, supported components with their own goals and needs. As discussed in our previous work [2], 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 [2].
- 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).

5 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.
- 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 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 [21, 14].

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 component's licenses. Our approach is based on Hohfeld's classic group of eight fundamental jural relations [16], of which we use right, duty, no-right, and privilege. We start with a tuple <actor, operation, action, object> for expressing a right or obligation. The actor is the "licensee" for all the licenses we have examined. The operation is one of the following: "may", "must", "must not", or "need not", with "may" and "need not" expressing rights and "must" and "must not" expressing obligations. 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.

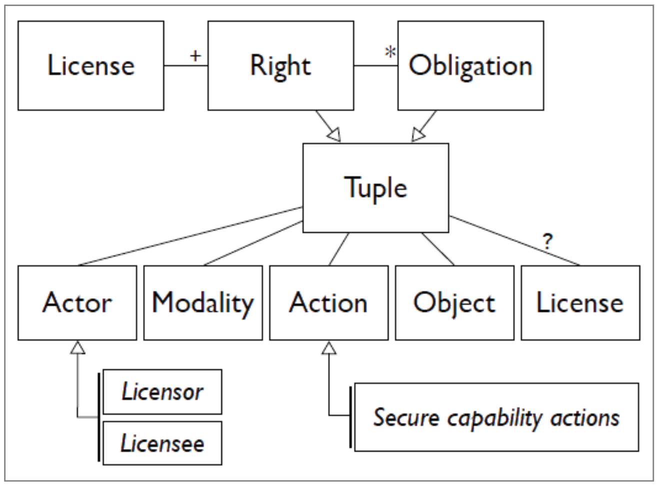


Figure 9: Security license meta-model

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.

6 Security License Architectures

Our security license model forms a basis for effective reasoning about licenses in the context of actual systems, and calculating the resulting rights and obligations. In order to do so, we need a certain amount of information about the system's configuration at design-time, build-time, and run-time deployment. The needed information comprises the license architecture, 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 guide/assess system design and evolution, using an automated environment of the kind that we have previously demonstrated [2, 3, 4, 5].

7 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 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 [3, 5], and are extending this approach to security licenses.

7.1 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 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 this 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) The right 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.

7.2 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.

8 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 program to access, manipulate, control, update, or evolve data, control signals, and accessible software system elements. Our proposed security license is influenced by IP licenses that are employed to specify property control and declared copyright freedoms/restrictions, such as those for OSS components subject to licenses like the GPLv2, MPL, LGPL, or others. These IP licenses as information structures often pre-exist to facilitate their widespread use, dissemination, and common interpretation. Further, the choice of which IP license to choose or assign to a software component results from a trade-off analysis typically performed by the components producers, rather than the system integrators or consumers, as a way to protect or propagate the obligations and rights to use, evolve, and redistribute the updated component's open source code.

The security licenses we propose may or not necessarily exist prior to their specification and assignment to a given OA system. Similarly, we may anticipate or expect that generic security licenses will emerge and be assigned by software component producers, as they have for OSS components, though no such security licenses from producers yet exist. However, one follow-on goal we seek to address is whether and how best to specify security licenses for different types of software elements or components so that it becomes possible to semi-automatically specify the security license for a given component or composed OA system through the reuse and instantiation of security requirement templates. This idea is somewhat similar to the license templates and taxonomy that is employed by the Creative Commons for non-software intellectual property like online art or new media content (cf. http://creativecommons.org/licenses/). In this regard, it may be possible to develop a technique and supporting computational environment whereby system integrators or consumers can conveniently specify the security requirements they seek (e.g. fill out online security requirements forms), while the environment interprets these specifications to generate operational security capabilities that can be guard the entry and exit of data or control information from the appropriate containment vessel that encapsulates the corresponding system element. Consequently, this is a topic for further study and investigation.

Next, one might wonder why it is not simply desirable to have maximum system security under all circumstances. When considering the alternative run-time system composition variants shown in Figures 5, 6, and 7, it appears 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 smallscale software elements (e.g., Cshell inter-application integration or run-time scripts). Similarly, the different security containment layouts may affect system performance, ease of evolutionary update, and associated level of security administration. But these again all represent trade-offs in the desire to achieve affordable, practical, and evermore robust and testable secure software component/system capabilities build-time and run-time. Thus, we take the position that it is better to provide the ability to specify and analyze the security requirements of different software elements at designtime, as well as specify and analyze the security capabilities at 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 that follows from our approach to semantically modeling and analyzing OA systems that are subject to software security licenses. More specifically, how our approach and emerging results might shed light on software systems whose architectures articulate a software product line.

Accordingly, organizing and developing software product lines (SPLs) relies on the development and use of explicit software architectures [7, 10]. 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 a SPL consisting of possible different system instantiations composed from similar components instances (e.g., different but equivalent Web browsers, word processors, email, calendaring applications, relational database management systems).

Third, if the SPL is based on an OA that integrates software components from multiple vendors or OSS components that are subject to heterogeneous security licenses (i.e., those that may possible conflict with one another), then we have the situation analogous to what we have presented in this paper. So secure SPL concepts are compatible with secure OA systems that are composed from heterogeneously security licensed components.

9 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 obligations and rights for how they may access data or control information, as well as how the may update or evolve system elements. These security license rights and obligations thus 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 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 grants #N00244-10-1-0038, #N00244-10-1-0077, and #N00244-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #0808783 from the U.S. National Science Foundation. No review, approval, nor endorsement implied.

References

[1] T. A. Alspaugh and A. I. Anton. Scenario support for effective requirements. Information and Software Technology, 50(3):198–220, Feb. 2008.

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

[3] T. A. Alspaugh, H. U. Asuncion, and W. Scacchi. Intellectual property rights requirements for heterogeneously-licensed systems. In 17th IEEE International Requirements Engineering Conference (RE'09), pages 24–33, Aug. 31–Sept. 4 2009.

[4] T. A. Alspaugh, H. U. Asuncion, and W. Scacchi. The Challenge of Heterogeneously Licensed Systems in Open Architecture Software Ecosystems, In S. Jansen, S. Brinkkemper, and M. Cusumano (Eds.), Software Ecosystems, Edward Elgar Publishing, (to appear, 2012).

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

[6] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

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

[8] T. D. Breaux and A. I. Anton. Analyzing goal semantics for rights, permissions, and obligations. In RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering, pages 177–188, 2005.

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

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

[11] N. Falliere, L. O Murchu, and E. Chien. W32.Stuxnet dossier, Version 1.4, February 2011, http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

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

[13] D. Firesmith. Specifying reusable security requirements. Journal of Object Technology, 3(1):61– 75, Jan.–Feb. 2004.

[14] R. Fontana, B. M. Kuhn, E. Moglen, M. Norwood, D. B. Ravicher, K. Sandler, J   Vasile, and A. Williamson. A Legal Issues Primer for Open Source and Free Software Projects. Software Freedom Law Center, 2008.

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

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

[17] F. Kuhl, R. Weatherly, and J. Dahmann. Creating computer simulation systems: an introduction to the high level architecture. Prentice Hall, 1999.

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

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

[20] P. Oreizy. Open Architecture Software: A Flexible Approach to Decentralized Software Evolution. PhD thesis, University of California, Irvine, 2000.

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

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

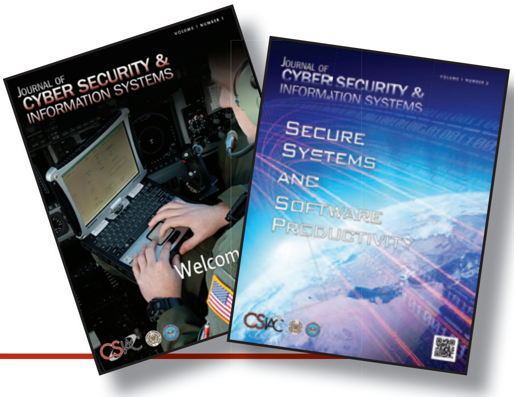
[23] S. S. Yau and Z. Chen. A framework for specifying and managing security requirements in collaborative systems. In Third International Conference on Autonomic and Trusted Computing (ATC 2006), pages 500–510, 2006.

About the Authors

Walt Scacchi is a senior research scientist and research faculty member at the Institute for Software Research, University of California, Irvine. He received a Ph.D. 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, 21-28 May 2011, Honolulu, HI.

Thomas Alspaugh is adjunct professor of Computer Science at Georgetown University, and 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.

Cyber Security & Information Systems Information Analysis Center



Call for Papers for Publication

The Cyber Security and Information Systems Information Analysis Center (CSIAC) - https://thecsiac.com, is one of eight Department of Defense Information Analysis Centers (IACs) sponsored by the Defense Technical Information Center (DTIC) - http://www.dtic.mil/dtic/.

CSIAC has been formed as the consolidation of three legacy IAC's – the Information Assurance Technology Assurance Center (IATAC), the Data and Analysis Center for Software (DACS), and the Modeling and Simulation Information Analysis Center (MSIAC) – along with the addition of the new technical domain of Knowledge Management and Information Sharing. CSIAC is chartered to leverage best practices and expertise from government, industry, and academia on Cyber Security and Information Technology. CSIAC's mission is to provide DoD a central point of access for Information Assurance and Cyber Security to include emerging technologies in system vulnerabilities, R&D, models, and analysis to support the development and implementation of effective defense against information warfare attacks.

CSIAC publishes the quarterly *Journal of Cyber Security and Information Systems*, focusing on scientific and technical research & development, methods and processes, policies and standards, security, reliability, quality, and lessons learned case histories. The latest issue may be viewed or downloaded at https://www.thecsiac.com/journal/welcome-csiac.

During the calendar year 2013 CSIAC will be accepting articles submitted by the professional community for consideration.

Articles in the areas of ***Information Assurance, Software Engineering, Knowledge Management, Information Sharing,*** and ***Modeling & Simulation*** may be submitted.

CSIAC will review articles and assist candidate authors in creating the final draft if the article is selected for publication. However, we cannot guarantee publication within a fixed time frame. Note that CSIAC does not pay for articles published.

To Submit an Article

Drafts may be emailed to Journal@thecsiac.com.

Preferred Formats:

- Articles must be submitted electronically
- MS-Word, or Open Office equivalent

Size Guidelines:

- Minimum of 1,500 – 2,000 words (3-4 typed pages using Times New Roman 12 pt font)
- Maximum of 12 pages, double column, including references
- Authors have latitude to adjust the size as necessary to communicate their message

Images:

- Graphics and Images are encouraged.
- Print quality, 200 or better DPI. JPG or PNG format preferred.

For the full Article Submission Policy, see page 38 of this journal.

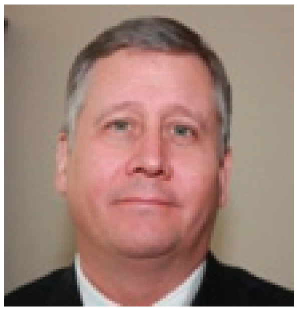
CSIAC Webinars

Each month CSIAC presents live webinars, free of charge, via WebEx. Previous webinars are archived on vimeo.com and can be viewed at any time. Here are a few of our recent webinars. Upcoming webinars will be highlighted on the CSIAC home page.

COST ESTIMATION OF AGILE PROJECTS

Agile has become a popular development methodology in software and systems development in recent years, but how do we tailor our estimation processes to this new methodology? Traditional methods do not apply in terms of project sizing and planning. How can we find an accurate point of comparison with industry trends? Presented by industry veteran Larry Putnam, Jr., QSM takes you through the basic steps on how to customize the estimation process to Agile.

Lawrence H. Putnam



Lawrence H. Putnam, Jr., Co-CEO Larry has 23 years of experience using the Putnam-SLIM Methodology. He has participated in hundreds of estimation and oversight service engagements, and is responsible for product management of the SLIM-Suite of measurement tools and customer care programs. Larry has delivered numerous speeches at conferences on software estimation and measurement, and has trained – over a five-year period – more than 1,000 software professionals on industry best practice measurement, estimation and control techniques and in the use of the SLIM-Suite.

→ **Watch & Listen: http://vimeo.com/21448188**

CLOUD NINE, ARE WE THERE YET?

In 1961 at the MIT Centennial, John McCarthy opined "if computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility.... the computer utility could become the basis of a new and important industry" [1]. In 2006, Amazon Web Services was launched providing computing on a utility basis. Since that time the notion of cloud computing has been emerging and evolving. Cloud computing is a paradigm that makes the notion of utility computing a reality. Instead of Information Technology (IT) organizations investing in all of the hardware, software and infrastructure necessary to meet their business needs, cloud computing makes access to hardware, software and infrastructure available through the internet, generally utilizing a pay for use model. Basically cloud computing allows an organization to adopt a different economic model for meeting IT needs by reducing capital investments and increasing operational investments, a model which is likely to offer cost savings to many organizations.

This webinar introduces the concept of cloud computing and discusses the potential benefits for a business as well as those things which could be barriers to adoption. It examines the types of applications where cloud computing is an efficient cost effective solution and the types of applications where its use could be problematic or costly. Several examples of successful cloud implementations are presented and discussed.

Arlene F. Minkiewicz

Arlene F. Minkiewicz is the Chief Scientist at PRICE Systems, LLC. In this role, she leads the cost research activity for the entire suite of cost estimating products that PRICE provides. Ms. Minkiewicz has more than 27 years of experience with PRICE building cost models. Her recent accomplishments include the development of cost estimating models for complex systems and systems of systems as well as research focused on the costs and benefits associated with migration to Service Oriented Architectures. Minkiewicz has published many articles on software measurement and estimation and frequently presents her research at industry forums.

→ **Watch & Listen: http://vimeo.com/25096379**

MEASURE QUALITY AND QUANTIFY RELIABILITY OF CRITICAL SOFTWARE

Software in critical applications must operate safely and reliably. Since testing can only show the presence of defects, but not their absence, how can engineers be confident that their software is robust? Using static code analysis with formal methods, it is possible to prove the absence of certain run-time errors in source code. By using these techniques software teams are able to quantify where software may or may not fail.

This webinar will introduce advanced verification techniques for software intended for critical applications. Through demonstrations, case-studies, and examples, attendees will learn how to develop and verify high quality software, use workflows and tools that improve software quality and reliability, and prove to certification authorities that certain types of defects can never occur.

Jay Abraham

Jay Abraham is currently a Technical Manager at The MathWorks. His area of expertise is in software tools for the verification of critical embedded applications. He has 21 years software and hardware design experience. Jay began his career as a microprocessor designer at IBM followed by engineering and design positions at hardware, software tools, and embedded operating systems companies such as Magma Design Automation and Wind River Systems. He has held vice-chairmanships in IEEE standards committees and has presented at conferences and publications such as the Design Automation Conference, Embedded Systems Conference, International System Safety Conference, and Systems and Software Technology Conference. Jay has a MS in Computer Engineering from Syracuse University and a BS in Electrical Engineering from Boston University.

→ **Watch & Listen: http://vimeo.com/25831986**

MODEL DRIVEN DEVELOPMENT

The idea of generating code from models has been implemented in a variety of ways with relative success. Most commercially available tools are UML-based, but the amount of code automatically produced from UML diagrams by those tools is relatively small compared to that of a fully functional application. Consequently, with most UML-based tools, hand-coding is still the most significant phase in the process of applications development and is therefore one of the areas in which most of the effort is put by development teams.

Project sizing and cost estimation are, along with the implementation itself, perhaps the two most crucial issues that a project manager has to face. Project managers have to estimate costs and allocate resources to the development of a system before it is built, so they tend to rely on their experience in having built similar systems in the past, and some even use rules of thumb (e.g. estimate the size of the system to be built based on the number of database tables it will have to deal with).

This presentation/demonstration will describe an MDA-based approach that provides developers with a technology that not only lets them define and, unlike most tools, automatically produce fully functional applications but that also automatically sizes (using the International Function Point Users Group (IFPUG) functional sizing methodology) the systems to be developed even before the first line of code is generated. The resulting functional size is then used for calculating the cost of code generation. Thus, this is actually a testimony of how one software service provider has developed its business model around the "cost per delivered functional unit" construct.

Whereas UML (among other diagram types) provides for Static (class diagram) & Dynamic (state transition diagram) object models, an ideal solution would also incorporate a Functional (services, methods, operations, calculations, etc.) and Presentation Model (user interface definitions). If the Model can instantiate a complete system (client, 100% business logic, dbms etc.) then dramatic improvements in productivity (Function Points/day) can be realized.

Juan Carlos Molina

Juan Carlos Molina has a BS in Computer Science from the Technical University of Valencia (Spain) and is the Research and Development Manager for Integranova. Having worked as applications developer for mainframe and information systems for several years, in 1998 he joined the OO-Method research group at the Technical University as academic researcher to develop a model compiler to transform object-oriented conceptual models into Visual Basic applications. In 1999, he was hired by CARE Technologies, a spin-off resulting from the reasearch group, now Integranova, where he coordinates the R&D division that develops Integranova Model Execution System, a set of tools which fully support an MDA-based, conceptual model-centric software development approach. In the last years he has coordinated and participated in the development of model validators, formal grammars, model compilers, sizing tools and metamodeling tools.

Greg Bishop

Greg Bishop is a graduate of the US Air Force Academy (1972) and served as an Intelligence Officer from 1972-1978 primarily in Southeast Asia and Germany . He went to work at Hewlett-Packard in sales in 1978 and retired in 2010 working mainly in the Federal sector. He is now the North American Director of Sales for Integranova and lives in Littleton, CO.

→ **Watch & Listen: http://vimeo.com/26961405**

SECURING SYSTEMS THROUGH SOFTWARE RELIABILITY ENGINEERING

Software reliability engineering (SRE) represents a well-established set of techniques for specification and assessment of dependability of software-based systems. Application of these techniques to security concerns could provide further helpful assistance for software assurance efforts.

Key aspects of SRE include establishing quantitative reliability targets, constructing usage profiles of the operational system, and conducting statistically based testing to predict system reliability.

Security analysis would build on the success of SRE by establishing multiple quantitative targets including availability and risk exposure, using threat modeling to identify a variety of misuse cases, and fundamentally rethinking software reliability growth modeling.

This presentation will also introduce the "Roadmap to Dependability" project and invite participants to become active in developing or reviewing "Roadmap" products.

Taz Daughtrey

Taz Daughtrey is Senior Software Quality Scientist at Quanterion Solutions and teaches software engineering at James Madison University. He is a Fellow of the American Society for Quality, where he was the Founding Editor of their journal Software Quality Professional. Taz has extensive experience in the commercial and naval nuclear power industries as well as providing training and consulting throughout North America, Europe, and Japan. He is currently also a Director of the American Software Testing Qualifications Board.

→ **Watch & Listen: http://vimeo.com/28164978**

IT ESTIMATION 6 TRUTHS YOU MUST LEARN

Software estimation is both art and science. Many good cost software cost estimation models, strategies, tools and techniques exist and are useful. However, estimation for NON-SOFTWARE information technology intensive projects is a domain that is often characterized by ad-hoc cowboy practices yielding inconsistent performance. But why change and how could you? This Webinar presented by Galorath Incorporated provides valuable information on improving IT estimation performance. It will discuss experience and research that indicates that when an organization coalesces tribal knowledge into organizational intelligence performance improves. Following that, an overview of the characteristics and attributes of the SEER for IT tool and how they address crucial needs of the IT shop.

Karen McRitchie

Karen McRitchie is a VP of Development. Ms. McRitchie is responsible for the design, development, implementation and validation of the parametric estimation relationships found in the SEER™

estimation product line. Ms. McRitchie has participated in numerous estimation, data collection, and calibration efforts and has trained hundreds of cost analysts in the use, application, and calibration of SEER-SEM™ and SEER-H™. She has been active in the International Society of Parametric Analysts (ISPA) for many years was honored in 2002 with the Parametrician of the Year award.

→ **Watch & Listen: http://vimeo.com/30454475**

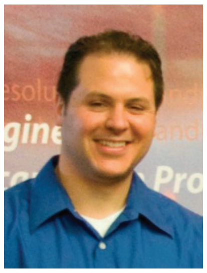
PARTITIONING COMPLEXITY: BREAKING DOWN WITHOUT BREAKING DOWN

Complex systems are, in a word, complex. The underlying complexity of a given problem is constant. Complexity can be hidden, but it cannot be eliminated. Consequently, one major objective should be aimed at effectively managing complexity through an appropriately layered partitioning approach. Beginning in the systems engineering domain (top-down design), it is necessary to conduct a (strictly human) process of defining and bounding the problem space such that the initial set of steps taken in the system specification design accurately reflect a solution to the actual problem at hand, and not a redefinition of the problem. With sufficient functional and non-functional requirements, the next steps further the top-down design in which the system is partitioned into compositional subsystems, and within each subsystem a set of cooperative related entities that carry individually assigned tasks. With the system specification in place, the other dimension of complexity that underlies the problem space is handled (bottom up) in the software engineering domain. As the SLOC (source lines of code) required for implementation of the functional requirements increases, the human effort required for handling various non-functional aspects such as performance, security, testability, readability, and reusability and therefore understandability and maintainability, can very quickly grow beyond manageable thresholds unless certain design principles are adhered too. Automation is necessary to assist with management of aspects including but not limited to testability and reliability. One major design objective should be to optimize the use of "abstraction" to achieve appropriate levels of functionality partitioning. In this way we localize complexity in order that the system remains manageable regardless of how large it becomes. To achieve this, the design approach must appropriately adopt a multi-paradigm philosophy; otherwise abstraction becomes a means to only hide complexity, and this is dangerous because it influences us to forget that the complexity still exists, and often creates the illusion that magic is happening! In this talk we discuss a system engineering philosophy and a software engineering methodology that combines object oriented and aspect oriented models.

Michael Weir

Michael Weir is a Senior Systems Analyst with Quanterion Solutions, Inc. He has over 30 years of experience in military R&D and operational deployment of communications and networking systems in the United States and overseas. Having seen the good, bad, and the ugly in the flow of systems development and deployment (or not) over three decades, Mr. Weir has a perspective on where the hard problems begin, and on how they evolve and consume resources that should be productive, but aren't. His current focus with complex systems is on effectively recognizing trust and identity and handling them properly across system and sub-system boundaries.

Mike Corley



Mike Corley is a Senior Software Architect for Quanterion Solutions, Inc. He is associated with the DACS (Data Analysis Center for Software) and the Air Force Research Laboratory (AFRL) in Rome. He has more than 10 years of professional experience providing technical R&D support to AFRL in areas including signal processing, computer/network and cyber security, and applications requiring complex software system architecture design and programming best practices. Mr. Corley is a recently admitted Ph.D. candidate at Syracuse University.

→ **Watch & Listen: http://vimeo.com/31872231**

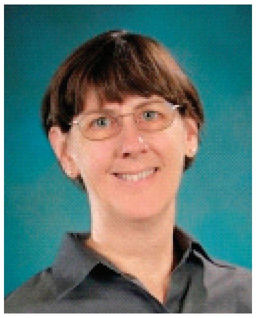
SYSTEM OF SYSTEMS CAPABILITY-TO-REQUIREMENTS ENGINEERING

Given an existing set of interconnected, independent systems, often referred to as a system of systems (SoS), one of the key activities according to the DoD Systems Engineering Guide for Systems of Systems is “translating SoS capability objectives into high-level SoS requirements”. Capability engineering starts with understanding the desired capability and identifying various options for achieving that capability. Initial capability engineering is typically done by assessing available resources and assets to identify existing functions from which the new capability can be composed, followed by a gap analysis for each alternative identified. Finally, each alternative is further evaluated in terms of capability performance, cost, and schedule, resulting in information that can be used to support the trade decision. This presentation:

- Provides additional guidance for translating capability objectives into requirements
- Defines SoS engineering (SoSE) methods, processes, and tools (MPTs) that might support this activity
- Illustrates how the SoSE MPTs would be used and integrated to support SoS engineering using Regional Area Crisis Response SoS (RACRS) example

While many of the techniques and methods described here are not new, they are used in ways tailored to support SoS and SoSE analyses and integrated together through a process to support capability-to-requirements engineering in a more rigorous, repeatable manner, resulting in meaningful information about alternatives that can be used to support a final decision on how the capability will be implemented. The MPTs described here are illustrated using the RACRS SoS. RACRS is a notional SoS that has been crafted to support SoSE research using actual systems in the public domain often employed to respond to regional crisis situations.

Jo Ann Lane



Jo Ann Lane is a research assistant professor at the University of Southern California Center for Systems and Software Engineering, conducting research in the areas of software engineering, systems engineering, and system of systems engineering (SoSE). She was a co-author of the 2008 Department of Defense Systems Engineering Guide for Systems of Systems. Current areas of research include SoSE processes, SoSE cost modeling, and SoS constituent system interoperability. Prior to her current work in academia, she was a key technical member of Science Applications International Corporation›s Software and Systems Integration Group for over 20 years, responsible for the development and integration of software-intensive systems and systems of systems. She received her PhD in systems engineering from the University of Southern California and her Master’s in computer science from San Diego State University.

→ Watch & Listen: http://vimeo.com/36080073

MODERN FILE AND DATA STRUCTURES FACILITATE DATA HIDING AND INFORMATION EXFILTRATION

The complexity and flexibility that is built into modern file and data structures provide opportunities for criminals or worse to hide, exfiltrate and covertly communicate data. This webinar will discuss file / data structure vulnerabilities and exploits. We will also discuss mitigating strategies that will help to close the gaps now and we will discuss future approaches to data and file structures that could systemically reduce the threat.

Chet Hosmer

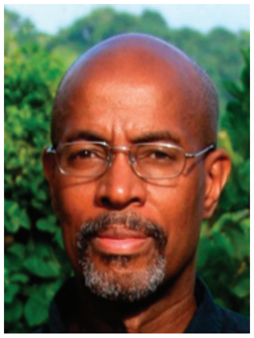
Chet Hosmer, Chief Scientist WetStone/Allen , has been researching and developing technology and training surrounding data hiding, steganography and watermarking for over a decade. He has made numerous appearances to discuss the threat steganography poses including National Public Radio’s Kojo Nnamdi show, ABC’s Primetime Thursday, NHK Japan, Crime TechTV and ABC News Australia. He has also been a frequent contributor to technical and news stories relating to steganography and has been interviewed and quoted by IEEE, The New York Times, The Washington Post, Government Computer News, Salon.com and Wired Magazine. Chet also delivers keynote and plenary talks on various cyber security related topics around the world every year.

→ Watch & Listen: http://vimeo.com/37661944

MODEL BASED SYSTEMS ENGINEERING: A SOLUTION TO COMPLEXITY OR JUST A COMPLEX SOLUTION?

Model Based Systems Engineering (MBSE) has been around for decades and has enjoyed a considerable amount of success and acceptance in industry and academia. However, MBSE is not without its challenges, particularly with respect to its practical application to large scale system development. As we research solutions to these open MBSE challenges, it is important for us to evaluate the state of MBSE based on how much system complexity it reduces relative to how much complexity it adds to the system development process itself. This interactive virtual panel features leading MBSE experts from industry, academia and the government discussing this and other issues. Using real-world experience, backed by data artifacts and experiments from the research and application domains documented in the SPRUCE portal and elsewhere, the panelists will present their points of view and wrap up with an evaluation of the state of the MBSE practice and actionable ideas that you can start implementing today.

Dr. Robert France



Dr. Robert France is a Full Professor in the Department of Computer Science at Colorado State University. His research interests are in the area of Software Engineering, in particular formal specification techniques, software modeling techniques, software product lines, and domain-specific modeling languages. He is an editor-in-chief of the Springer journal on Softw are and System Modeling (SoSyM), and is on the editorial board of IEEE Computer and the Journal on Software Testing, Verification, and Reliability. He is a past Steering Committee Chair of the MODELS/UML conference series, and is the PC Chair for MODELS 2012. He was also a member of the revision task forces for the UML 1.x standards. He was awarded the Ten Year Most Influential Paper award at MODELS in 2008.

→ **Watch & Listen: http://vimeo.com/38052031**

we like your feedback

At the CSIAC we are always pleased to hear from our journal readers. We are very interested in your suggestions, compliments, complaints, or questions. Please visit our website http://journal.thecsiac.com, and fill out the survey form. If you provide us with your contact information, we will be able to reach you to answer any questions.

Software Productivity Progress during the First Decade of the 21st Century: Quantifying Productivity

ByDonald J. Reifer

This paper summarizes the progress various industries have made in software productivity during the first decade of the 21st century. It begins by summarizing the productivity gains that occurred from 2000 and 2010 using data from 1,000 completed projects, none of which is over ten years old. Next, it categorizes the project data that were used to make further comparisons in terms of organizational demographics (size of engineering workforces for parent firms), degree of outsourcing, amount of contracting, life cycle stage (requirements, development and/or maintenance) and development paradigm employed (agile, evolutionary, iterative/incremental development, spiral or waterfall). Finally, the factors to which productivity growth seems to be most sensitive to are then identified by industry grouping and discussed as we look into the future to forecast what the next decade has in store for us.

Introduction

The topic of productivity continues to be a concern to those organizations developing software. The reason for this is that most of these groups have invested thousands upon thousands of dollars in processes, methods, tools and training aimed at reducing costs, increasing quality and improving operational effectiveness via productivity improvements. However, questions persist as to whether or not these investments have yielded positive returns. Let us take the case where a small software firm of 30 professionals brings in a new technology like agile methods [1, 2] as an example. When you tally the costs involved, the expenses for new software tools and agile classes alone will average $10,000 per employee when you consider that one full week of time will be needed for training. To achieve a payback of $300,000 a year, as shown in Table 1, this firm will have to achieve a productivity increase of 20% during the first year of the initiative exclusive of other costs. Most proponents would agree that achieving such a yield during the initial year of the move to agile would be highly unlikely due to large learning curves. A two year time window would be a much more reasonable assumption especially if other costs like mentoring and process changes (daily standup meeting, etc.) were assumed by the analysis.

	Year 1	Year 2	TOTAL
Current productivity (ESLOC/SM)[1]	100	110	
Work that can be achieved by 30 professionals (ESLOC)[2]	36K	39.6K	
Improvement (10%/year)	110	121	
Added work that can be achieved at 10% increase (ESLOC)	3,600	3,960	
Savings with 10% improvement ($50/ ESLOC)[3]	$180K	$198K	$378K
Improvement increased to 20% per year	120	144	
Added work that can be achieved at 20% increase (ESLOC)	7,200	12,240	
Savings with 20% improvement ($50/ ESLOC)	$360K	$612K	$972K

Table 1 - Notional Payback for Agile Initiative

Table 1 Notes

1. Assume that productivity increases 10% annually normally
2. ESLOC, as will be discussed later in the paper, takes into account reused and legacy code
3. The cost of $50/ESLOC was an average taken from the software tools domain for illustrative purposes only

The question that the example raises is how much productivity improvement can you expect to achieve realistically when making such major investments? In addition, what are the productivity norms that you would compare against and how are they computed? These are some of the questions that this paper was written to address.

2 Definition of Productivity

Software productivity is defined by IEEE Standard 1045-1992 [3] as the ratio of units of output generated to units of input used to generate them. The inputs typically consist of the effort expressed either in staff-months or staff-hours of effort expended to produce a given output. Outputs are most often represented in either source lines of code, function points or some proxy alternative (use case points, story points, feature points, web objects, etc.). Labor is typically scoped from the start of software requirements analysis through delivery i.e., the completion of software integration and test. It includes the following twelve related activities:

- Software requirements analysis
- Software architectural design
- Software implementation
- Software integration and test
- Software development test & evaluation
- Software task management
- Software documentation
- Software version control
- Software peer reviews
- Software quality engineering
- Liaison with other engineering groups
- Integrated product team participation

All directly charged labor is included in this tabulation. For example, labor hours for the leads who charge the project directly are included. Labor hours for executive management who charge overhead or other accounts are excluded as are charges made by the quality assurance staff who report to groups that are independent of the software engineering organization.

For outputs, we use equivalent new logical source lines of code (SLOC). This definition permits modified (including major modifications where changes are greater than 50 percent and minor modifications where alterations are less than 50 percent), reused (software that is designed for reuse) [4], auto-generated/generated code, and code carried over without modification from one build or increment to another to be equated to new source lines (i.e., this a separate category from modified, reused and generated code as it is defined in terms of entire components which are carried forward and used as is without modifications in a software build or delivery). It also allows function point sizing estimates to be related to SLOC counts using industry accepted conversion factors [5] where function points and their variants (feature points, web objects, etc.) are a specification rather than SLOC-based measure of software size.

The formula used to determine equivalent SLOCs, which is expressed as follows, builds on and extends work done by Dr. Barry Boehm for the popular COCOMO-II estimation model [4]:

Equivalent SLOC = New + (0.65) Major Modifications + (0.35) Minor Modifications + (0.15) Reused + (0.08) Auto-Generated/Generated + (0.02) Carry-Over

Many issues make counting such inputs and outputs hard [6]. To address these issues, a set of standard counting conventions must be developed and used. The conventions that we adopted for our analysis are based on those devised for the COCOMO-II cost model and are as follows:

- The scope of all projects starts with software requirements analysis and finishes with software delivery, i.e., completion of software development test & evaluation.
  - For commercial systems, the scope extends from approval of project startup until sell-off. In other words, the scope for applications systems development extends from product conception to sell-off to the customer.
  - For defense systems, the scope extends from the software requirements review until handoff to the system test for hardware and software integration and testing. It assumes that the software deliverables have successfully passed its software development test & evaluation (software selloff).
- Projects can employ any of a variety of methods and practices that range from simple to sophisticated.
- Effort includes all chargeable engineering, management and support labor in the numbers.
  - It includes software engineering (requirements, design, programming and test), task management and support personnel (data management, configuration management, network administration, etc.) who charge the project directly.
  - It typically does not include independent quality

assurance, system or operational test, and beta test personnel who charge their time to accounts other than software. It does include the software effort needed to support those who charge these accounts (liaison, meetings, integrated product team assignments, etc.).
  - As an example, the software effort includes version control. However, it does not include the software effort needed to assess the impact of a major change to the system if this is funded through a Program Office account and charged separately.
  - As another example, the software effort includes responding to a quality assurance group's independent assessment of their product if the quality group had a separate charge number which they charged to.
- The average number of hours per staff month was 152 (used to take holidays, vacation, sick leave, etc. into account).
- SLOC is defined by Florac and Carleton [7] to be logical source line of code using the conventions issued by the Software Engineering Institute (SEI) as altered by current practice. All SLOC counts adhere to these counting conventions with the exception of deleted lines which are included in the counts in the same manner as added lines.
- Function point sizes are defined using current International Function Point Users Group (IFPUG) standards.
- Function point sizes were converted to source lines of code (SLOC) using backfiring factors published by various experts as available on their web sites and in the public domain (e.g., see http://sunset.usc.edu/classes/cs510_2011/ECs_2011/ec39-Software%20Sizing.ppt, p. 30).
- Projects used many different life cycle models and methodologies. For example, web projects typically used some form of agile process and lightweight methods, while military projects used more classical waterfall-oriented or incremental processes and methods. Commercial projects used object-oriented methodology predominantly, while military projects used a broader mix of conventional and object-oriented approaches.
- Projects used a variety of programming languages. For example, web projects employed Java, Perl and Visual C while military projects used predominately C/C++.
- Most projects surveyed used more than one programming language on their projects. The cost per SLOC when provided is therefore provided for the predominate language. Some adjustments were made to combine different language generations.
- Dollars used to determine cost are assumed to be constant year 2010 dollars.
- The cost assumed per staff month (SM) of effort of $18,000 assumes an average labor mix and includes all direct labor costs plus applicable overhead, but not general & administrative (G&A) costs and profit. The mix assumes that the average staff experience across the team in the application domain was 5 to 8 years. It assumes that staff has the skills and experience needed with the application, methods and tools used to develop the products.
- Many of the organizations surveyed were trying to exploit commercial off-the-shelf packages, open source software and legacy systems (existing software developed previously whose heritage and quality varied greatly) to reduce the volume of work involved. Piecing the system together and making it work were a challenge.
- Most of the defense and government organizations surveyed were rated at CMMI Level 4 or greater, while most commercial firms were not [8]. For the most part, the processes that commercial organizations used were mostly ISO certified.

Productivity and Quality

Productivity can easily be manipulated by eliminating tasks within the scope identified and altering definitions. For example, including comments and/or blank lines in the definition of a source line of code (SLOC) can effectively increase size by 20 to 40 percent based on which programming language(s) is being employed. As another example, limiting scope to extend from detailed design through software testing results in productivity numbers that are 40 to 60 percent higher because they involve a smaller percentage of the total software development effort. As a final example, productivity numbers for software maintenance are much higher than the actuals because they do not include the entire effort within their scope. Because these maintenance numbers include only the numbers for the generating new releases, they appear to have higher productivity than they should [9]. In reality, they should also include the effort required to sustain the older versions of the software in the field within their scope. They should also address the effort that the maintenance staff expends to sustain the platforms and facilities to get a true picture of the actual productivity that the organization is achieving.

In addition, productivity can also be easily increased by manipulating quality. For example, increasing the defect density from our current norm of measurement of 1 to 3 to a rate of 3 to 10 defects/thousand equivalent source lines of code (EKSLOC) upon release of the software to the field can result in productivity increases of between 18 to 40 percent for a telecommunications application of between 100 and 250 EKSLOC. In many cases, such quality degradations occur inadvertently when the products are released to the field prematurely due to deadline pressures.

Productivity and Cost

Finally, many people confuse productivity improvement with cost reduction. Productivity and cost in terms of effort expended to generate a software product are separable issues because the factors that influence them are different. For example, the easiest way to increase productivity is to get staff to generate more output per unit of input using automation and tools. In contrast, costs can most easily be cut by outsourcing the work overseas to reduce labor costs.

You can validate the concept that productivity and cost are related but separable variables by realizing that you can increase both at the same time by being very productive generating the wrong thing. This phenomenon becomes apparent when your team generates the wrong product very efficiently because it failed to capture the right requirements.

3 Data and Data Bases Employed

In many people's minds, the key to determining productivity, cost and quality relationships is to basing your results on empirical data. Over the past 30 years, I have gathered data from over 100 software development shops via data exchange agreements. In exchange for their data, we provide them productivity norms classified by industrial sector and application domain. They use these norms to determine how well they are performing relative to their industry and competition. As part of the agreement, we sign legal documents that define how we will protect the data because the data is considered very proprietary. This makes sense because these firms do not want their competition to know what their current productivity is and what their actual cost and quality norms are. As a result, we can neither share the details of our databases nor the data contained within them. We can only share our analysis results.

Some have criticized this practice of presenting data without providing the details because they distrust results that they cannot see, replicate and validate. Others are happy to have benchmarks that they can compare their own baselines against even if there is no access to the underlying data used to determine the benchmarks. Because we believe some in the community can use our results, we have supplied it. However, we advise making an effort to validate these benchmarks against your own numbers. There will be critics who will question the source of the numbers especially if the benchmarks do not tell the story that the critics want to recite.

Data Bases

Table 2 summarizes the range of actual software productivity experienced during 2010 along with their range of size and defect densities for 1,000 projects within our databases by industrial sector and application domain for the 93 current organizations from which we currently gather data. Please note that the 2000 database [10] was characterized differently and as such some of the entries associated with it had to be adjusted for normalization purposes. In addition, the project tallies and size ranges in the Table pertain to the 2010 database, not the 2000 entries. Finally, note that the entries in the two databases can only be normalized and compared when they have a like scope and definitions.

The division between firms within the commercial and defense sectors is about 75:25. All of the data have been refreshed so that no data point is more than 10 years old in the 2010 database. In addition, Table 2 includes relatively large amount of new data in the 2010 database supplied by seven new organizations, all of which are commercial. The industrial sectors included are:

- ***Commercial*** – firms within this sector include those profit-making organizations that design, develop, field, maintain and sustain Information Technology (IT) products and services. In some cases, these organizations may manufacture and sell such systems commercially. In others, they may integrate and use products supplied by others to automate their systems and procedures (production control, finance and accounting, travel, etc.).
- ***Defense*** – firms within this sector include those profit-making organizations that design, develop, field, maintain and/or sustain systems used by the military for a full range of weapons system (fire control, mission planning, sensor data processing, situation awareness, etc.) and support (medical, information systems like billings, etc.). Such systems tend to be more complex than their commercial counterpart primarily because they involve safety and security issues.

For Table 2, defect density in the table refers to the number of defects per thousand lines of code (d/KSLOC) measured after delivery. KSLOC in this context is measured in equivalent lines of code to take modified, reused, generated and carry-over code into account as noted earlier in the COCOMO

II discussion. While there has been some work aimed at identifying how many defects developers should expect as they generate software products, these benchmarks vary a great deal as a function of process. However, these developmental defect rates can be used very effectively when the software process is controlled (i.e., as a function of process maturity) to predict the number of defects remaining in a build. This metric gives developers insight into whether they have tested enough assuming that they have set some goal for the numbers of defects remaining at delivery.

Sector	Application Domain	No. of Projects[1]	Size Range (EKSLOC)[2]	Productivity Range (ESLOC/SM)[3]	Defect Density (d/KESLOC)[4]
Commercial	Automation	65	45 to 325	225 to 407 (272)	4.1
(729)	Command and Control	73	35 to 3,875	78 to 265 (165)	0.75
	Information Tech[5]	106	30 to 4,580	229 to 522 (363)	4.3
	Medical[7]	35	45 to 1,125	238 to 433 (305)	2.5
	Scientific Systems	44	35 to 1,090	185 to 387 (231)	1.6
	Software Tools[8]	106	18 to 1,895	236 to 441 (332)	4.8
	Telecommunications	81	25 to 3,732	175 to 435 (321)	2.7
	Test Systems[7]	49	28 to 785	225 to 453 (323)	3.3
	Training/Simulation	35	45 to 2,130	243 to 477 (313)	4.1
	Web Business	135	20 to 655	255 to 665 (315)	6.8
Defense	Military – Airborne	52	18 to 2,330	68 to 197 (123)	0.3
(271)	Military – Ground	51	35 to 5,210	87 to 275 (215)	0.6
	Military – Info Tech[7,9]	59	25 to 1,755	173 to 497 (335)	3.9
	Military - Medical[7]	33	45 to 950	212 to 417 (285)	1.3
	Military – Missile	25	15 to 647	55 to 143 (98)	0.2
	Military – Space	31	22 to 1,750	83 to 198 (117)	0.3
	Military – Trainers[7]	20	45 to 2,250	172 to 498 (311)	1.0
TOTAL		1,000	15 to 5,210	55 to 665 (257)	2.65

Table 2 – Number of Projects by Domain and Industrial Sector (2010)

Table 2 Notes

1. Projects are software development activities that result in delivery of a product to a customer. Under certain circumstances, they can include increments when they are delivered to the customer for interim use as other increments are being developed.
2. Size in equivalent KSLOC uses the approach in Section 1 to address the following five categories of code: new, modified (segmented by major and minor changes), reused (designed for reuse), generated and carry-over (refactored for use as-is with no modifications; e.g., build 1 code used in build 2).
3. Range of productivity for these 2010 benchmarks have the average value denoted in parentheses. The definition of the life cycle scope and what labor is included in the computation is as noted in Section 1.
4. Refers to the average number of defects per thousand SLOC after delivery. These defect rates have held relatively constant across the decade because they have been established as goals of the development.
5. Was Data Processing in the 2000 benchmark article.
6. Most of the data collected for these benchmarks were submitted using function points. Function points were converted to lines of code so that measures reported could be normalized. Conversion factors were developed to take into account the mix of programming languages employed, reuse and other factors based on guidance provided by IFPUG (International Function Point Users Group).
7. These five applications domains are new for the 2010 benchmark.
8. Was Environments/Tools in 2000 benchmark article.
9. The root cause of productivity being substantially lower in the military-info tech domain than its commercial counterpart revolves around the amount of rigor and degree of governance applied, not the defect rates established as goals.

As summarized Figure 1, the net productivity improvement across the decade ranges from between 2 to 3 percent per year independent of whatever was viewed as the newest software salvation. Table 3 summarizes the degree of software productivity improvement experienced by sector and application domain using the norms established in 2000 that are noted in the Table and used as our basis for increases [10]. The reason the rate is lower than might be expected is that current requirements are tougher and software products that we are building are getting bigger and more complex. Based on these findings, the community needs to set more realistic goals as they continuously introduce technology to make it easier to develop our products (i.e., our make technology). However, even at such levels these modified productivity improvement goals are sufficient to expend funds on new technology especially when compelling technical and economic arguments are made to justify investments [11]. However, you will need to select the technologies carefully and take charge of the risk involved in making the systematic transition to its widespread use [12]. Else, the technology you try to use may rapidly overwhelm you and your chances of success will diminish accordingly.

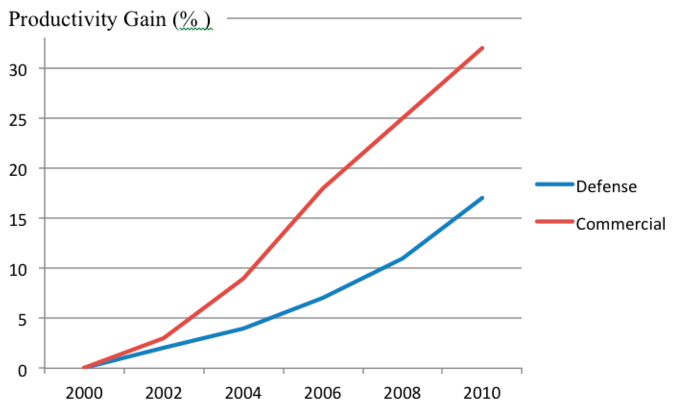


Figure 1 - Trends in Productivity Improvement by Sector during the last Decade

As shown in Table 3, commercial projects have experienced a 31.8% gain in software productivity over the decade, while defense projects improved about half of that at 17.1%. The contributing factors to these large differences in software productivity improvement during the last decade between these two sectors revolve around the software development and management practices used (including the degree of automation), reliability expectations, complexity of the product, and workforce influences. For example, defense firms have taken a more conservative approach when it comes to adopting agile methods because of the nature of their business which is governed by acquisition regulations and contract law.

Based on their underlying nature and use, defense systems tend to be larger, more complex and have higher reliability expectations than their commercial counterparts. The practices used during the decade have gotten leaner and more agile [2]. Automation has become widely used during development for tasks like refactoring [13] and testing [14]. During the decade, workforce issues have dominated as firms have strived to address the economic downturn by cutting their costs through out-sourcing and increasing their productivity through the use of automation and packaged software.

While these factors will not change during the next decade, the systems that we work on will. Systems will continue to become more and more distributed as will the workforces that will be used to develop them. Social media and new forms of collaboration will be developed to address the needs of the ever expanding global community developing software. System-of-system principles [15] will dominate the engineering solutions that will evolve that will exploit such new solutions as cloud computing, social networking and model-based software development. As a consequence of these actions, systems and software engineering will continue to converge especially when building software-intensive systems and new more agile and lean engineering methods will be developed to accelerate getting products to market at reasonable costs.

Sector	Application Domain	Productivity Range (ESLOC/SM) 2000 Database	Productivity Range (ESLOC/SM) 2010 Database	Percent Gain[1, 2] (Decade)
Commercial	Automation[3]	120 to 440 (225)	225 to 407 (272)	20.9%
(754)	Command & Control	95 to 330 (225)	78 to 265 (165)	+
	Info Tech[3,4]	165 to 500 (270)	229 to 522 (373)	38.2%
	Medical5	No data	238 to 433 (305)	N/A
	Scientific Systems	130 to 360 (195)	185 to 387 (231)	18.5%
	Software Tools[6]	143 to 610 (260)	236 to 441 (332)	27.7%
	Telecommunications	175 to 440 (250)	175 to 435 (321)	28.4%
	Test Systems[5]	No data	225 to 453 (323)	N/A
	Training/Simulation	143 to 780 (224)	243 to 477 (313)	39.7%
	Web Business[3]	190 to 975 (225)	255 to 665 (315)	40.0%
			Average gain across all domains - Commercial	31.8%
Defense	Military – Airborne	65 to 250 (105)	68 to 197 (123)	17.1%
(246)	Military – Ground	80 to 300 (195)	87 to 275 (215)	10.2%
	Military – Info Tech[5]	No data	173 to 497 (335)	N/A
	Military - Medical[5]	No data	212 to 417 (285)	N/A
	Military – Missile	52 TO 165 (85)	55 to 143 (98)	15.3%
	Military – Space	45 to 175 (90)	83 to 198 (117)	30.0%
Military – Trainers[5]		No data	172 to 498 (311)	N/A
			Average gain across all domains - Defense	17.1%

Table 3 – Productivity Improvement by Domain and Industrial Sector from 2000 to 2010

Table 3 Notes

+ The types of command and control projects included in the 2000 and 2010 databases were defined differently in that the former included several instances that were reclassified as automation and process control. The number of process control entries in the 2000 database brought the average much higher. If these projects were eliminated, the productivity would have been on the order of 135 ESLOC/SM. However, we do not believe the results are comparable because different technology was used.

1. Gain within an application domain is calculated as (2010 - 2000 entries)/2000 entry. Overall gain is computed as the weighted average across all domains for which we have entries in both databases.
2. Because the data were refreshed to keep it current, few of the projects included in the 2000 database appear in the 2010 database.
3. Average productivity in 2000 was changed in the Table due to normalization.
4. Was Data Processing in the 2000 benchmark article.
5. These five applications domains are new for the 2010 benchmark.
6. Was Environments/Tools in 2000 benchmark article.

Application domains employed are characterized in Table 4 to provide a more complete description of the types of projects within our databases. It is important to note that developing application domain specific numbers allows us to improve our statistical accuracy.

Domain	Characterization
Automation	This domain supports applications like those used in an auto/truck assembly plant. Here automated conveyer belts transport the vehicle through a series of finishing stages. Robots assemble the vehicle and specialized diagnostic tools perform quality checks. All of these actions are under the control of supervisory system with a man-in-the-loop to contend with reliability issues. Process control applications are included as well like those in oil pipeline operations.
Command & Control	This supports applications like those used for network control and switching and for displaying information to users. An example system would be a smart house and many of the applications that reside in it. Such software is command-driven and highly interactive. In some applications like determining and displaying situational awareness during operations, the software operates in real-time.
Information Technology	This domain is a catch-all for many types of information systems including applications in the following nine NAICS codes: • Agricultural, forestry and fishing • Mining (coal, metals, oil and gas extraction, etc.) • Construction • Manufacturing (chemicals, food, metals, petroleum, rubber, tobacco, etc.) • Transportation, communications, electric, gas and sanitary services • Wholesale trade • Retail trade (food stores, apparel, home furnishings, etc.) • Services (auto repair, education, entertainment, legal, travel, etc.) • Public administration (justice, taxation, legislation, etc.)
Medical	This domain supports systems developed to support patients in clinics, hospitals and in care facilities including in the home. They provide for doctor, dental, vision and prescription drug care. Medical systems are for the most part currently client-server systems that provide centralized access to a range of information including that used for wellness, diagnosis and patient care.
Scientific Systems	This domain supports number-crunching applications like weather or seismic processing. In such systems, large numbers of computers are used in parallel to perform mathematical calculations. For example, seismic systems do oil prospecting by taking samples and filtering lots of noisy geological data over and over, again and again. Weather modeling and forecasting is another application that falls into this category.
Software Tools	This domain supports software development and maintenance operations with specialized packages and systems aimed at enhancing productivity and cutting costs. A tool is a program that developers use to create, debug, maintain and otherwise support other programs and applications. They can be provided as standalone packages or integrated collections of tools called environments.
Telecommunications	This domain supports a wide range of developments for broadband, land line, microwave, radio, satellite, and wireless applications. This domain includes a wide range of applications because the software supports phones, faxes, wireless devices, telegraph, transmission systems, switching systems, satellite systems, local exchanges and a host of other types of equipment including broadband networks, virtual and private branch exchanges (PBX).
Test Systems	This domain supports a variety of automatic test equipment designed to diagnose problems in both hardware and software, including components like integrated circuit cards. These systems control the execution of tests, the comparison of predicted to actual outcomes, the setting up of test preconditions, and other test control and reporting functions.

Table 4 - Characterization of Application Domains (continued on next page)

Training/Simulation	This domain supports a variety of training devices, from games to full-blown system simulators. The large trainers/simulators typically are being built to train commercial pilots, pipeline operators, and other personnel in the use and care of complicated equipment and systems. Currently, medium-sized simulators are being developed for applications like e-learning and e-commerce.
Web Business	This domain supports a full range of applications ranging from e-travel reservation systems to integrated web-based business suite that do everything including financials, inventory and e-commerce (ordering, sales, etc.). Such systems range in size from individual to corporate web sites tied to the Internet and private networks. They include wikis and blogs and work groups. They support instant messaging, twitter and other social networking technologies.
Military - Airborne	This domain supports embedded applications aboard fighters, helicopters and other types of aircraft including drones and unmanned aerial vehicles. Military airborne applications range the gamut from real-time sensor processing systems like terrain-following radars to more mode-based mission management systems.
Military - Ground	This domain includes systems that range from artillery to air defense to tanks to manned and unmanned combat vehicles to reconnaissance platforms. The characteristics of these projects vary greatly as does the size. Some integrate information to perform tasks like tracking troop movement so that their location is known. In contrast, others line unmanned systems include robots that roam the battlefield gathering intelligence as they search for enemy troops.
Military - Information Technology	This domain is a catch-all for many types of information systems being developed and maintained by and for the military. While these systems share similar features with their commercial counterparts, they are less broad and designed to support the military's needs for current and accurate information.
Military - Medical	This domain supports systems developed to support active duty personnel in the field, in hospitals and in care facilities including in the home. They provide doctor, dental, vision and drugs at military hospitals and clinics on base. They provide first aid and combat related services at home and in the field.
Military - Missile	This domain supports typically embedded air-to-air, air-to-ground, ship-to-shore, ship-to-ship, ground-to-space, air-to-space and space-to-space applications - both air (missile) and ground (launcher). Systems also in this category also include more modern weapons like ground-based anti-missile missiles and exoatmospheric kill vehicles, guided projectiles and directed energy systems.
Military - Space	This domain supports a wide variety of systems that operate both in space and on the ground to perform missions ranging from satellite control to weather mapping. Such systems operate in real-time in space autonomously or under the control of a ground station. Many types of applications are supported.
Military - Trainers	This domain supports a variety of realistic rehearsal and training systems. Soldiers should train as they should fight. In response, training systems are built to use mission equipment to provide students with as near an environment as they would expect to experience operationally. The look, feel and sounds of the mission are replicated as are the operational procedures, including those related to exercising emergency procedures.

Table 4 - Characterization of Application Domains (continued from previous page)

4 More about the Data

The data that these findings and forecasts were predicated upon was compiled from 93 leading organizations in many industries across the United States. As shown in Figure 2, the size of the engineering workforces involved in supplying the 1,000 project data points used ranged from twenty to thousands of engineers using the classifications that follow:

- Small – less than one hundred engineering employees.
- Medium – between one hundred and five hundred engineering employees.
- Large – over five hundred engineering employees.

Counts used do not assume that employees were in a single location. However, all of the locations surveyed as part of the effort were within the United States. In some cases, contractors that were working as members of software teams were included within the counts because these personnel fulfilled roles that employees normally performed during the development. In these cases, it seemed easier for the organization involved to get money than staff authorizations.

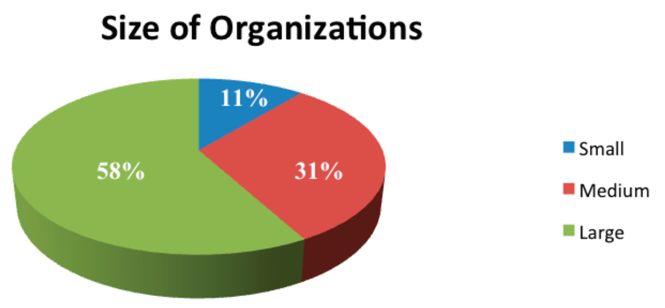


Figure 2 - Size of Engineering Workforces

For the most part, the staff working defense projects was more experienced than those working commercial ones. So were staff members performing maintenance rather than development tasks; i.e., the maintenance teams had 8 years versus 4 years of average experience. Figure 3 highlights the comparison between commercial and defense projects by summarizing the average years of experience taken from our 1,000 project database.

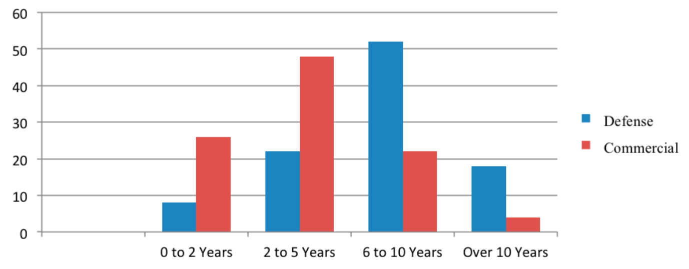


Figure 3 - Average Workforce Experience

The software development paradigm used to develop software has changed radically during the past decade within the commercial sector. A software paradigm consists of a framework for modeling software development consisting of a set of activities, methods, practices, reviews and transformations programmers use to develop software and associated products. As shown in Figure 4, commercial and defense firms differed greatly in the paradigms that they used (see below for definitions) to get their software development projects completed:

- ***Waterfall Process*** – refers to a sequential software development process in which progress is seen as flowing steadily downwards (like a waterfall) through a series of phases like analysis, design, implementation, testing, qualification, delivery and maintenance. Providing feedback between phases and using well defined milestones to gate the transitions between stages are viewed as the keys to success with this paradigm [16].
- ***Iterative or Incremental Process*** – refers to a cyclical software development process in which progress is made in iterations like builds or releases after the initial planning and analysis is completed. A build is defined for these purposes as a group of programs that are integrated, tested and qualified before delivery to the client. Deployment is made iteratively with cyclical interactions focusing on addressing feedback into builds or releases [17].
- ***Spiral*** – refers to a cyclical software development process that combines elements of both design and prototyping-in-stages to combine advantages of both the waterfall and incremental process using a single methodology. Early focus is placed on reducing requirements volatility in hope that this will speed the later stages of development [18].
- ***Evolutionary*** – refers to a cyclical software development process where feedback is passed from one evolution of the system (could be either an increment or spiral) to another using a well-defined and disciplined set of practices [19].
- ***Agile*** – refers to a group of modern software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. Several agile varieties ranging from those that focus on defining tests first to those that start by defining requirements employing user stories [20].
- ***Hybrid*** – some unique combination of any of the paradigms listed above whose aim is to take advantage of them when developing software.

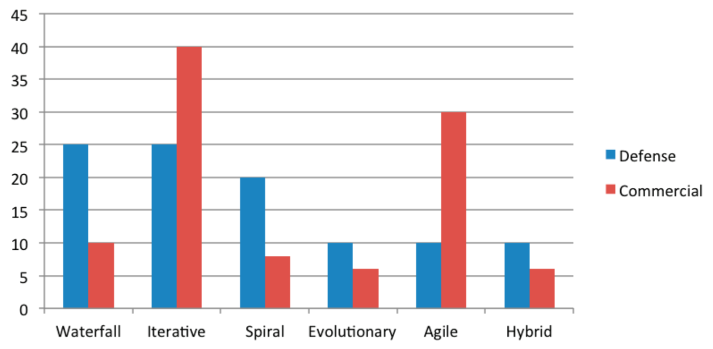


Figure 4 - Distribution of Software Paradigm Employed During Development by Sector

The totals in Figure 4 did not originally sum to one hundred percent in some cases. The reason for this is that some projects in our databases used multiple paradigms. During the early stages, they developed the software incrementally. Then, as the system matured and was fielded, they switched to a waterfall approach because the risk was acceptable. In other cases, parts of the system like servers were developed incrementally, while other parts like clients and web applications used agile methods for their development paradigm. For ease of use, we have normalized all both the defense and commercial entries to total one hundred percent each.

Many people have asked me what the differences in productivity were when using agile and other paradigms. My answer has been that it depends. The reason for this is most agile projects that were captured in our database in the 2000 to 2008 timeframe were small and it was hard to compare them with others that were much larger in scale. However, we started to get larger project after 2008 and have included Figure 5 to illustrate the comparisons for both the defense (18 projects, 6 each methodology) and commercial (48 projects, 16 each methodology) sectors across all applications domains for projects that were less than 250 EKSLOC. It seems that moving smartly to agile seems cost-effective based on the benefits derived through increased software productivity.

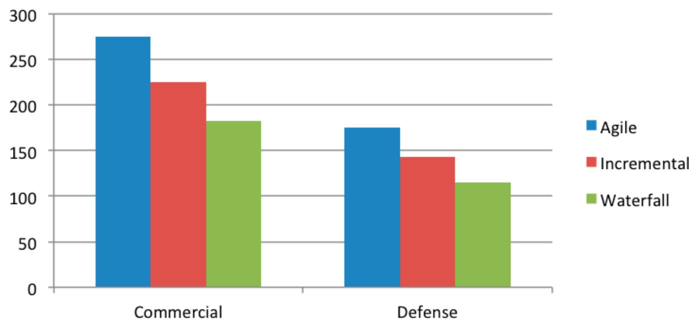


Figure 5 - Productivity Comparison based on Development Paradigm

Figure 5 Notes

- Software productivity reported for paradigms using definitions and scope in Part 1 of this article using SLOC/PM.
- Agile projects are for the most part being done using Scrum methodology as it seems to scale for use on larger projects [2].

Another of the factors that influenced software productivity was degree of in-sourcing and out-sourcing. Due to the economic downturn during the past decade, commercial firms have increased out-sourcing software work overseas. In other cases, they use H-1B visas to bring qualified personnel into the United States to do specific software jobs at wages less than the prevailing salaries in the marketplace. Costs are definitely cut by such practices, sometimes by as much as two to ten. However, cultural issues, communications and increased management burden can cause resulting productivity to be less than expected for teams residing in the United States. However, most firms using out-sourcing have addressed these issues and the practice is viewed as cost-effective based on the prevalence of the approach as illustrated in Figure 6.

It should be noted that many defense organizations have resisted out-sourcing primarily because of security issues. They use United States citizens exclusively to do the work because of its sensitivity. In other cases, both commercial and defense firms have resorted to in-sourcing work to address their staffing needs and reduce costs (i.e., hired contractors to work in-house as members of their teams). Off-loading work to residents (both citizens and sometimes green card holders) reduces management and communications burdens and permits productivity to rise naturally. Even though in-sourcing only lowers the labor costs by a factor of 20 to 50 percent, it tends to improve teamwork and productivity thereby making the practice cost-effective for many of firms involved in our data collection.

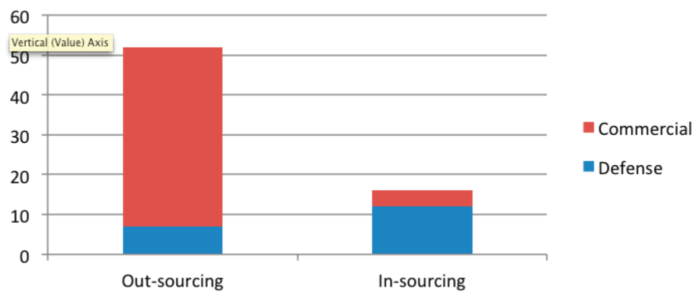


Figure 6 - In-Sourcing versus Out-Sourcing by Sector

5 Factor Analysis

As we mentioned earlier in this paper, productivity improvement is sensitive to a number of factors. The factors around which most of the variation in productivity revolves by sector and their impacts are summarized in Table 1. It is interesting to note that several of these factors have changed during the decade. The most prominent of these has been growth in size and complexity of the products and the relaxation of process improvement mandates. The simplest explanation for the relapse of the process mandate is that the firms that embraced the movement have for the most part achieved their goal of achieving high levels of maturity using either the CMMI or ISO model as their basis for evaluation. The other prevalent explanation is that firms have moved to agile because they felt that process improvement methodologies were too rigid and some balance between discipline and agility was needed [21].

Factor	2000 Database		2010 Database	
	Commercial	Defense	Commercial	Defense
Software development and management practices	Focus on ISO processes	Focus on CMMI processes	Move to agile methods	Focus continues on CMMI
	Size mostly moderate	Size small to moderate	Size gets bigger and bigger	Size gets bigger and bigger
	Requirements simple/ moderate	Requirements difficult	Requirements moderate/hard	Requirements more difficult
	Automation moderate	Automation moderate	Automation gets higher	Automation gets higher
	3rd and 4th GEN languages	Ada dies. Return to C/C++	4th and 5th GEN languages	3rd and 4th GEN languages
	Rate of progress reporting	Metrics-based management	Agile metrics	Earned value reporting
	Interdisciplinary teams	Integrated product teams	Social networking	Integrated product teams
Reliability expectations	Low to moderate	Moderate to very high	Low to moderate	Moderate to very high
Product complexity	Low to moderate	Moderate to very high	Low to moderate	Moderate to very high
	Client-server architecture	Real-time processing	Broadcast-subscriber	Real-time processing
	Mega data centers	Mega data centers	Movement to clouds	Movement to clouds
	Distributed systems	Distributed systems	Net-centric systems	Systems of systems
Workforce influences	Turnover high	Turnover moderate	Turnover high	Turnover moderate
	Young workforce	Middle aged workforce	Young workforce	Aged workforce
	Internet revolution	Internet revolution	Internet a way of life	Internet a security worry
	Shops mostly centralized	Shops mostly centralized	Shops mostly decentralized	Shops mostly centralized
	Starting to outsource	Starting to insource	Outsourcing normal	Insourcing normal

Table 5 - Factors that Influence Productivity by Sector by Year

As shown in Table 5, the continued advance of technology has had a profound impact on how we get the work associated with software projects done during the past decade. For example, the use of the Internet has facilitated the ability of geographically-dispersed teams to work together in virtual space to build products. Along with the good, the use of such technology also created problems. For example, working in virtual space has amplified the need for new forms of collaboration and tighter network security. In contrast, the advent of the use of social networking altered how our teams communicate with each other, clients and collaborators, both actual and potential. Finally, security in general has become a bigger issue as we store information in the clouds and share it across virtual space.

6 D ata Validation

Many people have questioned the validity of my databases over the years mainly because they could not independently review the data entries. As anyone who has dealt with data knows, firms treat their productivity, cost and quality actuals as proprietary. Any leakage could create damage especially if they fell into the hands of competitors who used them to win competitive bids. As a consequence, those who receive data must protect it. To release it requires supplier approval. In our case, this is an almost impossible task because it means that we would need to solicit the approval of as many 93 organizations before we could make it made public.

Of course, we validate the data as we receive it to ensure that it conforms to the conventions that we have set for its definition. If there are holes or inconsistencies in the data, we interact with the supplier to resolve the issues we have found prior to entering it into our databases. Once a new dataset is formed, we look for outliers and test it statistically. We also perform hypothesis testing and look to determine the statistical errors.

With the data tightly held, how would you go about validating our conclusions? The easiest answer to these questions is by comparison against your and others' results. While some productivity data has been made available by others like Capers Jones [22], the only productivity databases that I know that are maintained and kept up-to-date are those sold by the DACS (Data & Analysis Center for Software) [23] and International Software Benchmarking Standards Group (ISBSG) [24]. The DACS database has recently been upgraded by the University of Southern California to contain more current results. Figure 7 illustrates how the results reported in this paper compare to similar findings taken from these two sources. Again, the individual data are not available because of the need to maintain privacy. As the Figure illustrates, our findings seem reasonable when you make the effort to compare like data with like data.

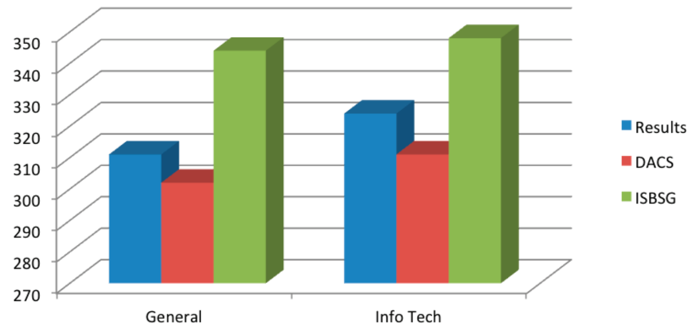


Figure 7 - Comparison of Productivity (SLOC/SM) Findings with DACS and ISBSG Benchmarks

Figure 7 Notes

- Because each of the benchmarks has different scopes, we had to normalize the life cycle phases and labor categories included.
- We made some assumptions about what to include because application domains were defined differently in each of the databases.
- We also converted function point size to source lines of code (SLOC) to ensure all size entries used the same basis.

7 Summary and Conclusions

In summary, the past decade has seen considerable improvements in the technology that most software organizations use to architect, develop, test, manage and maintain their products, both in the defense and commercial sectors. These improvements have in turn sparked software productivity increases that I believe are considerable. While some would be disappointed and say that the magnitude of these increases is not high enough for them, such gains are sufficient for anyone to use to justify major investments in new software techniques, tools and technology. For example, a mere 2 percent gain each year in productivity improvements in a firm that employs 1,000 software engineers justifies millions of dollars in expenditures. I am personally excited about the results and am hopeful that they will continue and accelerate.

I would like to take this opportunity to thank those in the community who have supported my productivity, cost and quality benchmarking activities over the years. They have provided the funding and data that have made it possible for me to be able to provide these results over the years. I would specifically like to thank Mr. Peter McLoone of Lockheed Martin and the DACS staff for their insightful inputs and suggestions for improving this manuscript.

Finally, in conclusion, I believe that putting these productivity results in the public domain is important because they provide the community with an empirical basis for comparison. Such comparisons will in turn help those running software organizations and projects to set realistic expectations. I encourage others who have such empirical results to make it public as well. This would enable the community to agree on numbers that everyone could and should use for the community's benefit.

References

[1] Cockburn, A., ***Agile Software Development***, Addison-Wesley, Upper Saddle River, NJ, 2002.

[2] Cohn, M., ***Succeeding with Agile***, Addison-Wesley, Upper Saddle River, NJ, 2010.

[3] IEEE Computer Society, ***IEEE Standard for Software Productivity Measurement***, IEEE Std 1045-1992, September 17, 1992.

[4] Boehm, B.W., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Reifer, D. and Steece, B., ***Software Cost Estimation with COCOMO II***, Prentice Hall, Upper Saddle River, NJ, 2000.

[5] Capers Jones published the original table that backfired function points to lines of code in the 1990's. These tables are now for sale at http://www.spr.com/programming-languages-table.html.

[6] Park, R, ***Software Size Measurement: A Framework for Counting Source Statement***, CMU/SEI-92-20, Software Engineering Institute, Pittsburgh, PA, 1992.

[7] Florac, W.A. and Carleton, A.D. (Editors), ***Measuring the Software Process***, Addison-Wesley, 1999.

[8] NDIA CMMI® Working Group, ***CMMI® for Small Business***, November 2010 (see http://www.ndia.org/Divisions/Divisions/SystemsEngineering/Documents/Committees/CMMI%20Working%20Group/CMMI4SmallBusiness_CMMIWG.pdf).

[9] Reifer, D.J., ***Software Maintenance Success Recipes***, CRC Press, Boca Raton, FL, 2011.

[10] Reifer, D.J., "Let the Numbers Do the Talking," ***Crosstalk***, March 2002, pp. 4-8.

[11] Reifer, D.J., ***Making the Software Business Case: Improvement by the Numbers***, Addison-Wesley, Upper Saddle River, NJ, 2001.

[12] Reifer, D.J., ***Software Change Management: Case Studies and Practical Advice***, Microsoft Press, Redmond, WA, 2011.

[13] Fields, J., Harvie, S., Fowler, M. and Beck, K., ***Refactoring: Ruby Edition***, Addison-Wesley, 2009.

[14] Kung, D.C., Hsia, P. and Gao, J., ***Testing Object-Oriented Software (Practitioners)***, Wiley, IEEE Computer Society Press, 1998.

[15] Jamshidi, M., ***Systems of Systems Engineering: Principles and Applications***, CRC Press, 2008.

[16] Royce, W.W., "Managing the Development of Large Software Systems," ***Proceedings Wescon***, IEEE, August 1970, pp. 1-9.

[17] Larman, C. and Basili, V., "Iterative and Incremental Development: A Brief History," ***Computer***, IEEE, June 2003, pp. 2-11.

[18] Boehm, B. W., "A Spiral Model of Software Development and Enhancement," ***Computer***, IEEE, May 1988, pp. 61-72.

[19] May, E.L. and Zimmer, B. A, "The Evolutionary Development Model for Software," ***Hewlett Packard Journal***, August 1996, available at the following web site: http://www.hpl.hp.com/hpjournal/96aug/aug96a4.pdf.

[20] The Agile Alliance web site provides broad coverage for the methodology at http://www.agilealliance.org/.

[21] Reifer, D. J., ***Software Change Management: Case Studies and Practical Advice***, Microsoft Press, 2011.

[22] Jones, C., ***Software Assessments, Benchmarks and Best Practices***, Addison-Wesley, Upper Saddle River, NJ, 2000.

[23] Information about the Data & Analysis Center for Software (DACS) is available at http://www.thedacs.com/.

[24] Information about the International Software Benchmarking Standards Group (ISBSG) is available at http://www.isbsg.org/.

Article Submission Policy

The CSIAC Journal is a quarterly journal focusing on scientific and technical research & development, methods and processes, policies and standards, security, reliability, quality, and lessons learned case histories. CSIAC accepts articles submitted by the professional community for consideration. CSIAC will review articles and assist candidate authors in creating the final draft if the article is selected for publication. However, we cannot guarantee publication within a fixed time frame.

Note that CSIAC does not pay for articles published.

AUTHOR BIOS AND CONTACT INFORMATION

When you submit your article to CSIAC, you also need to submit a brief bio, which is printed at the end of your article. Additionally, CSIAC requests that you provide contact information (email and/or phone and/or web address), which is also published with your article so that readers may follow up with you. You also need to send CSIAC your preferred mailing address for receipt of the Journal in printed format. All authors receive 5 complementary copies of the Journal issue in which their article appears and are automatically registered to receive future issues of Journal. Up to 20 additional copies may be requested by the author at no cost.

COPYRIGHT:

Submittal of an original and previously unpublished article constitutes a transfer of ownership for First Publication Rights for a period of ninety days following publication. After this ninety day period full copyright ownership returns to the author. CSIAC always grants permission to reprint or distribute the article once published, as long as attribution is provided for CSIAC as the publisher and the Journal issue in which the article appeared is cited. The primary reason for CSIAC holding the copyright is to insure that the same article is not published simultaneously in other trade journals. The Journal enjoys a reputation of outstanding quality and value. We distribute the Journal to more than 30,000 registered CSIAC patrons free of charge and we publish it on our website where thousands of viewers read the articles each week.

FOR INVITED AUTHORS:

CSIAC typically allocates the author one month to prepare an initial draft. Then, upon receipt of an initial draft, CSIAC reviews the article and works with the author to create a final draft; we allow 2 to 3 weeks for this process. CSIAC expects to have a final draft of the article ready for publication no later than 2 months after the author accepts our initial invitation.

For some issues CSIAC has a Guest Editor (because of their expertise) who conducts most of the communication with other authors. If you have been invited by a Guest Editor, you should

PREFERRED FORMATS:

- Articles must be submitted electronically.
- MS-Word, or Open Office equivalent (something that can be edited by CSIAC)

SIZE GUIDELINES:

- Minimum of 1,500 – 2,000 words (3-4 typed pages using Times New Roman 12 pt font) Maximum of 12 pages
- Authors have latitude to adjust the size as necessary to communicate their message

IMAGES:

- Graphics and Images are encouraged.
- Print quality, 200 or better DPI. JPG or PNG format preferred

Note: Please embed the graphic images into your article to clarify where they should go but send the graphics as separate files when you submit the final draft of the article. This makes it easier should the graphics need to be changed or resized.

CONTACT INFORMATION:

CSIAC
100 Seymour Road Suite C102
Utica, NY 13502
Phone: (800) 214-7921
Fax: 315-351-4209

John Dingman, Managing Editor
Phone: (315) 351-4222
Email: jdingman@quanterion.com

Tom McGibbon, CSIAC Director
Phone: (315) 351-4203
Email: tmcgibbon@quanterion.com

ABOUT THE JOURNAL OF CYBER SECURITY AND INFORMATION SYSTEMS

ABOUT THIS PUBLICATION

The Journal of Cyber Security and Information Systems is published quarterly by the Cyber Security and Information Systems Information Analysis Center (CSIAC). The CSIAC is a DoD sponsored Information Analysis Center (IAC), administratively managed by the Defense Technical Information Center (DTIC). The CSIAC is technically managed by Air Force Research Laboratory in Rome, NY and operated by Quanterion Solutions Incorporated in Utica, NY.

John Dingman
Managing Editor
Quanterion Solutions, CSIAC

Thomas McGibbon
CSIAC Director
Quanterion Solutions, CSIAC

Shelley Howard
Graphic Designer
Quanterion Solutions, CSIAC

Paul Engelhart
CSIAC COR
Air Force Research Lab

COVER DESIGN

Shelley Howard
Graphic Designer
Quanterion Solutions, CSIAC

Distribution Statement:
Unclassified and Unlimited

CSIAC
100 Seymour Road
Utica, NY 13502-1348
Phone: 800-214-7921
Fax: 315-351-4209
E-mail: info@thecsiac.com
URL: http://www.thecsiac.com/

ARTICLE REPRODUCTION

Images and information presented in these articles may be reproduced as long as the following message is noted:

"This article was originally published in the Journal of Cyber Security and Information Systems Vol.1, No.1 October 2012."

In addition to this print message, we ask that you notify CSIAC regarding any document that references any article appearing in the *CSIAC Journal.*

Requests for copies of the referenced journal may be submitted to the following address:

Cyber Security and Information Systems
100 Seymour Road
Utica, NY 13502-1348

Phone: 800-214-7921
Fax: 315-351-4209
E-mail: info@thecsiac.com

An archive of past newsletters is available at **https://journal.thecsiac.com.**

Cyber Security and Information Systems
Information Analysis Center
100 Seymour Road
Suite C-102
Utica, NY 13502

Return Service Requested

PRSRT STD
U.S. Postage
P A I D
Permit #566
UTICA, NY

Journal of Cyber Security and Information Systems – February 2013
Secure Systems and Software Productivity

— IN THIS ISSUE —