2007-04-01

Putting Teeth into Open Architectures: Infrastructure for Reducing the Need for Retesting

Valdis Berzins

NPS-AM-07-018

EXCERPT FROM THE

PROCEEDINGS

OF THE

FOURTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM WEDNESDAY SESSIONS

Putting Teeth into Open Architectures: Infrastructure for Reducing the Need for Retesting

Published: 30 April 2007

by

Valdis Berzins, and

Manuel Rodríguez, Naval Postgraduate School

4th Annual Acquisition Research Symposium of the Naval Postgraduate School:

Acquisition Research: Creating Synergy for Informed Change

May 16-17, 2007

Prepared for: Naval Postgraduate School, Monterey, California 93943

Acquisition Research program
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

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

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

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

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

Conference Website:
www.researchsymposium.org

Proceedings of the Annual Acquisition Research Program

The following article is taken as an excerpt from the proceedings of the annual Acquisition Research Program. This annual event showcases the research projects funded through the Acquisition Research Program at the Graduate School of Business and Public Policy at the Naval Postgraduate School. Featuring keynote speakers, plenary panels, multiple panel sessions, a student research poster show and social events, the Annual Acquisition Research Symposium offers a candid environment where high-ranking Department of Defense (DoD) officials, industry officials, accomplished faculty and military students are encouraged to collaborate on finding applicable solutions to the challenges facing acquisition policies and processes within the DoD today. By jointly and publicly questioning the norms of industry and academia, the resulting research benefits from myriad perspectives and collaborations which can identify better solutions and practices in acquisition, contract, financial, logistics and program management.

For further information regarding the Acquisition Research Program, electronic copies of additional research, or to learn more about becoming a sponsor, please visit our program website at:

www.acquistionresearch.org

For further information on or to register for the next Acquisition Research Symposium during the third week of May, please visit our conference website at:

www.researchsymposium.org

THIS PAGE INTENTIONALLY LEFT BLANK

Putting Teeth into Open Architectures: Infrastructure for Reducing the Need for Retesting

Valdis Berzins
Naval Postgraduate School
Monterey, CA 93943, USA
E-mail: berzins@nps.edu

Manuel Rodríguez
Naval Postgraduate School
Monterey, CA 93943, USA
E-mail: mrodrigu@nps.edu

Abstract

The Navy is currently implementing the open-architecture framework for developing joint interoperable systems that adapt and exploit open-system design principles and architectures. This raises concerns about how to practically achieve dependability in software-intensive systems with many possible configurations when: 1) the actual configuration of the system is subject to frequent and possibly rapid change, and 2) the environment of typical reusable subsystems is variable and unpredictable. Our preliminary investigations indicate that current methods for achieving dependability in open architectures are insufficient. Conventional methods for testing are suited for stovepipe systems and depend strongly on the assumptions that the environment of a typical system is fixed and known in detail to the quality-assurance team at test and evaluation time. This paper outlines new approaches to quality assurance and testing that are better suited for providing affordable reliability in open architectures, and explains some of the additional technical features that an Open Architecture must have in order to become a Dependable Open Architecture.

Introduction

The Navy's Open Architecture (OA) is defined to be a multi-faceted strategy providing a framework for developing joint interoperable systems that adapt and exploit open-system design principles and architectures (DAU, 2007b). The objective of supporting adaptable systems has significant implications for quality assurance. OA approaches often involve: (i) a public, non-proprietary architecture that can accept plug-in components and be transparent to changes (e.g., the system should continue to work if selected components or connectors are replaced by different components or connectors), and (ii) an architecture whose purpose is to make explicit the common interfaces (e.g., POSIX, CORBA, etc.). Main goals of Navy's OA include minimizing total cost of ownership, increasing competition, achieving reuse, optimizing systems, and developing systems that support evolution.

This paper explores some test and evaluation implications, outlines an approach for providing affordable quality assurance in the kind of dynamic environment that open architectures are intended to accommodate, and evaluates the current state of some technologies that support the new approach.

The Navy's requirements to implement OA are set forth in several Department of Defense (DoD) and Department of Navy (DoN, 2004, August 5) policy documents (e.g., "The

Defense Acquisition System" (DoD, 2003, May 12), "Guidance Regarding Modular Open Systems Approach (MOSA) Implementation" (DoD, 2004, April 5), "Naval Open Architecture Scope and Responsibilities" (DoN, 2004, August 5), etc.). In the past the Navy has acquired systems that, although they performed their functions and tasks exceedingly well, were unique in their designs and engineering. Indeed, they required unique parts, equipment, and services to support them, were supported by a limited number of suppliers and became unaffordable to maintain. There are numerous instances, moreover, in which a system or platform was scrapped rather than upgraded or modernized because the cost to do so had become prohibitive. Test and evaluation account for an appreciable part of the cost for system upgrades. This paper explores how open architecture principles can be extended and applied to reduce these costs and to make Navy systems more agile.

Business issues are pushing the Navy to shift its development processes towards an open-architecture paradigm. In an era of strenuous competition for dollars, the Navy is continuously challenged with budget decisions. Inflexible acquisition strategies lock the Navy into single systems and vendors that limit the service's options for competition and innovation. Limited competition impedes innovation, while OA provides options for greater competition and inclusion of innovators. Cost of procured systems is due to maintenance as well as development expense. Stovepiped processes lead to acquisition of systems across the Navy with duplicated capabilities. For example, every ship (class) had a unique combat system. Currently, limited asset reuse takes place across the enterprise without open architectures. However, there are few enterprise processes to foster integration in a legacy environment. To achieve rapid fielding of new technology and capability for the Fleet, the Navy's business model has to change from the classic acquisition system to a process that supports Rapid Capability Insertion. Open architecture meets those needs by shortening cycle-times for getting capability to the warfighter when needed. The use of modular systems to facilitate technology refreshment and obsolescence mitigation is a key aspect of OA. Increased competition and innovation are possible through changed business practices enabled by OA.

Many technical issues are also motivating the Navy's change towards open architectures:

- Procurement of monolithic systems using legacy processes produces incompatible systems that are not interoperable.
- Software closely coupled (integral) to the computing hardware platforms is not reusable.
- Special-use code and modules that cannot be reused across the Navy are artifacts of the legacy approach to systems acquisition.
- Proliferation (and resulting lifecycle cost growth) of hardware and software baselines results from upgrade processes in closed systems.

Consequently, there has been much attention to cultural issues and acquisition policies to facilitate adoption of an open architecture paradigm for Navy systems.

This paper addresses a complementary effort to identify current weaknesses and gaps in the state of the knowledge with respect to assuring reliability of DoD/DoN systems developed according to open-systems principles, and to develop or adapt new methods for overcoming those weaknesses so they can be used in Navy open architectures. We are

studying weaknesses in current best practices with respect to the context identified above, and are performing research to extend and develop methods to overcome those weaknesses.

Our preliminary investigations indicate that current methods for achieving dependability in open architectures are insufficient. The main problem is how to practically achieve dependability in software-intensive systems with many possible configurations when the actual configuration of the system is subject to frequent and possibly rapid change, and the environment of typical reusable subsystems is variable (used in many platforms) and unpredictable (mission-dependent). This is a major problem for practical development because real development projects depend heavily on software testing, which is strongly context-dependent. Conventional methods for testing depend strongly on the assumptions that the environment of a typical system is fixed and known in detail to the quality-assurance team at test and evaluation time. These assumptions are quite reasonable for stovepipe systems but are not valid for open architectures. A component in an open architecture should be reusable not only across current classes of ships but also across future platforms that are yet to be designed—those that belong to different services, and perhaps even to coalition partners. This set of contexts is very large in practice, is open-ended, and cannot even in principle be known in detail to the test and evaluation team.

This paper outlines new approaches to quality assurance and testing that are better suited for providing affordable reliability in open architectures, and explains some of the additional technical features that an Open Architecture must have in order to become a Dependable Open Architecture, i.e., one that can support reuse and rapid reconfiguration via module swapping (without compromising reliability) while remaining economically viable at the level of individual systems and reducing total ownership cost for the enterprise. This requires linking the architecture with: 1) specific dependability requirements, 2) certifiable technical standards for each interaction path, 3) specialized types of testing, as well as combining that testing with other kinds of computer-aided quality-assurance methods. The paper explains the concepts behind the approach and why it is expected to work as claimed.

Navy's Vision of Open Architecture

The Navy Open Architecture (Navy OA) is a Navy initiative for a multi-faceted strategy providing a framework for developing joint interoperable systems that adapt and exploit open-system design principles and architectures (DAU, 2007a, DAU, 2007b). This is a systems design approach consistent with several governmental concepts and initiatives, such as the Open Architecture Computing Environment (OACE) (Naval Sea Systems Command, 2007), FORCEnet (FORCEnet, 2007a), and the Modular Open Systems Approach (MOSA) (Open Systems Joint Task Force, 2007). OACE seeks to ease the test and evaluation burden by limiting hardware choices to certain approved possibilities. FORCEnet is an operational concept that can benefit from realization of OA goals for its implementation. MOSA is a joint-acquisition approach that shares many of the goals of the Navy's OA effort.

The OACE (NSWCDD, 2004, August 23a, NSWCDD, 2004, August 23b) aims to implement open specifications for interfaces, services and supporting formats. It enables software components to work across a range of systems and interoperate with other software components on local and remote systems. Thus, the OACE framework includes a set of principles, processes, and best practices. The OACE is a surface-Navy approach to setting technical standards for shipboard systems. It shares many of the objectives of OA,

but does not address business processes that deal with those objectives, and does not apply to submarines, aircraft or C4I systems. The OACE consists of a set of documentation describing an infrastructure of technologies supported by a reference architecture. This infrastructure includes cable plant, cabinets, network components, processors, operating systems, adaptation and distribution middleware, frameworks, resource management, common services (e.g., system server applications such as web servers), etc. As an example, Figure 1 shows the reference OA defined by the OACE.

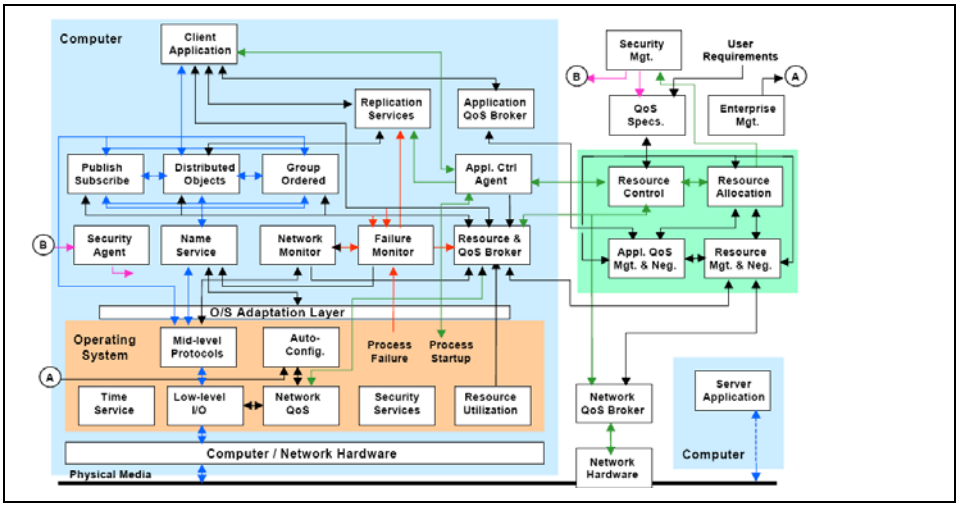

Figure 1. The OACE's Open Architecture Computing Environment
(extracted from NSWCDD, 2004, August 23a)

The OACE also defines guidance and strategies for fault tolerance, scalability, portability, real-time performance, system composition, system test & certification, and selection of standards (e.g., POSIX, CORBA, etc.). The OACE will allow the Navy to introduce and change out commercial technology to maximize affordability and performance goals.

FORCEnet is the operational construct and architectural framework for Naval Warfare in the Information Age to integrate warriors, sensors, networks, command and control, systems, platforms, and weapons into a networked, distributed combat force, scalable across the spectrum of conflict from seabed to space and sea to land (FORCEnet, 2007a). FORCEnet is, thus, the future implementation of the Network Centric Warfare in the Navy, and is the Navy's primary effort to integrate multiple architecture and standards efforts. Research efforts demonstrated that across the Navy Enterprise, FORCEnet viability, affordability and sustenance necessitates an architecture that is in full compliance with OA technology, systems and standards. The development and embedding of OA within FORCEnet will enable a superior, adaptive, "plug and fight" capability for the modern warfighter of today and tomorrow. Figure 2 presents the system interface view of FORCEnet.

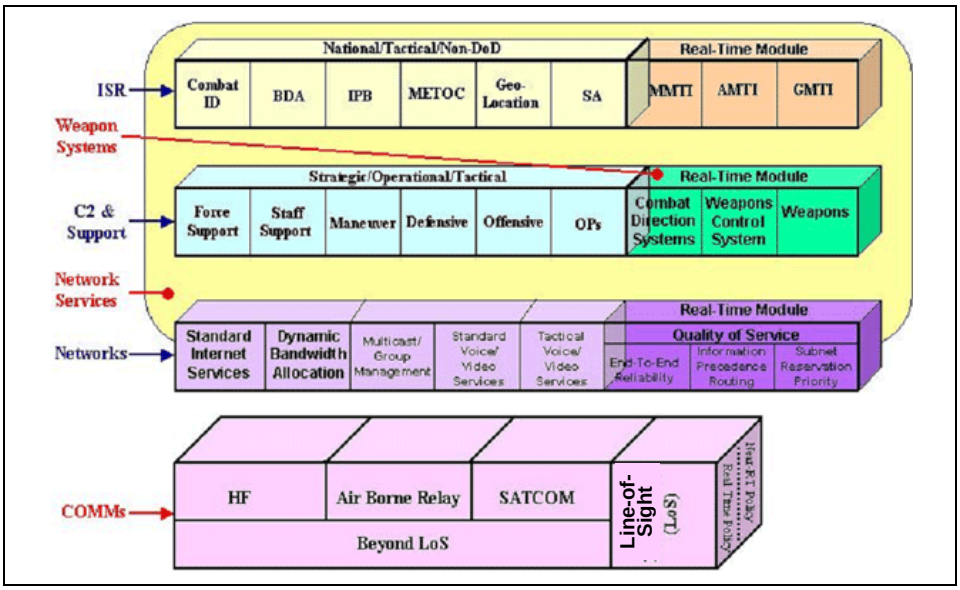

Figure 2. FORCEnet's System Interface Description
(extracted from FORCEnet, 2007b)

The Naval Open Architecture Enterprise Team (OAET) is currently spearheading an OA/FORCEnet Risk Reduction Experimentation effort to minimize the risk of delivering interoperable products (Shannon, 2006). This effort is in its early stages and has recently completed its first cycle. An example of a project enabling the integration of OA into FORCEnet is the "Open Architecture as an Enabler for FORCEnet" project (Deering et al., 2006, September). It concentrates on implementing network-centric military operations with specific threat-engagement scenarios (i.e., sensed threats to available weapons). These concepts are applied to the FORCEnet OA Domain Model using legacy and future warfare/Navy systems based on OA concepts. An analysis exposed potential functional boundary limitations in the current OA Domain model, and a revised model has been proposed.

The Modular Open Systems Approach (MOSA) (Open Systems Joint Task Force, 2007) is both a business and technical strategy for developing new systems or modernizing existing ones (see Figure 3).

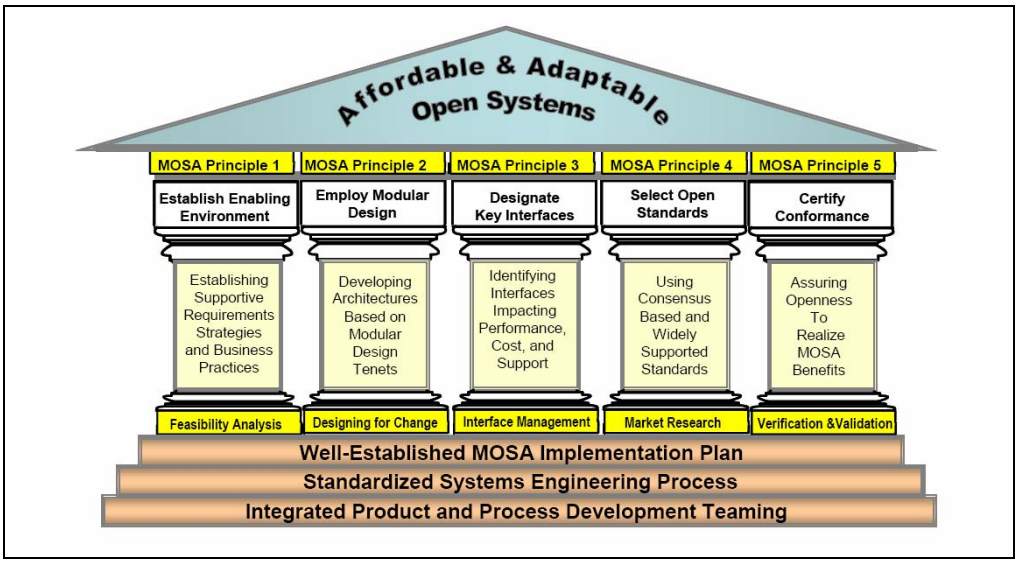

Figure 3. MOSA's Fundamental Building Blocks
(extracted from Flowers & Azani, 2004)

As a business strategy, the MOSA enables program teams to build, upgrade and support systems more quickly and affordably. This can be achieved through the use of commercial products from multiple sources and by leveraging the commercial-sector investment in new technology and products. The technical portion of the MOSA addresses a system design that is modular, has well-defined interfaces, is designed for change and, to the extent possible, makes use of commonly used industry standards for key interfaces. This system design is best accomplished using collaborative engineering based on sound systems engineering processes. Adherence to MOSA allows for developing DoD systems that account for the growing asymmetrical threats, unprecedented rate of technological change, and requirements for joint warfighting capabilities. The Navy's OA is closely related to MOSA. OA is a more specific extension of generalized MOSA principles. Naval OA applies to computer-intensive National Security Systems as defined in the Clinger Cohen Act, while MOSA has broader applicability, e.g., including mechanical systems.

The successful implementation of OA principles in the Navy may bring multiple benefits from both business and technical viewpoints to the Navy and other DoN/DoD organizations. Business benefits include: (i) enterprise-wide plans based on a cost/capability analysis of programs that address capability, affordability, and stabilization, (ii) flexible acquisition strategies and contracts that enable the Navy to reuse software, easily upgrade systems, and share data throughout the enterprise, (iii) streamlined investments in similar capabilities, (iv) increased competition to foster innovation and leverage technology upgrades, and (v) established enterprise processes and governance to foster integration. On the other hand, an efficient implementation of OA principles yields many technical benefits, including: (i) layered and modular open architectures that address portability, maintainability, interoperability, upgradeability and long-term supportability, (ii) modular, open designs consisting of components that are self-contained elements with well-defined interfaces, (iii) maximum use of commercial standards and commodity "commercial off-the-shelf" (COTS) products, and (iv) systems that continuously conform with Information Assurance (IA) requirements and monitor technology developments for IA improvements.

Figure 4 below presents a synthesis of OA benefits.

Performance	• Continuous competition yeilds best of breed applications • Focus on warfighting priorities
Schedule	• System integration of OA compliant software happens quickly • Rapid update deliveries driven by use operational cycles
Cost Avoidance Mechanisms	• Software - develop once, use often, upgrade as required • Hardware - Use high-volume COTS products at optimum price points • Training systems use same tactical applications and COTS hardware
Design for Maintenance Free Operating Periods	• Install adequate processing power to support "fail-over" without maintenance • Schedule replacement with improved COTS vice maintaining old hardware • Reduce maintenance training required • Consolidate Development and Operational Testing for reused applications
Risk Reduction	• Field new applications only when mature • Do not force the last ounce of performance • Deploy less (but still better than existing) performance or wait until next update

Figure 4. Open Architecture Benefits
(extracted from DAU, 2007C)

The Navy still needs to complete carrying out the necessary business and technical changes to achieve the stated OA goals. Well-known technical changes include the need for continuing the transition to COTS-based computing plants in modular architectures, the development of an OA Enterprise component library capability to facilitate market research and reuse of components, the alignment of standards among the domains and the alignment of standards to the DoD Information Technology Standards Registry (DISR). This paper identifies additional technical changes related to test and evaluation.

Difficulties in Testing Systems with Open Architectures

The Navy has emphasized improving its business and organizational processes, structure and expertise over technical matters. The Navy is currently able to deliver open architecture-based systems. However, known methods for achieving dependability with OA are expensive and not clearly understood. The Navy's current approach to system testing is not well matched to the needs of open environments. It is too expensive; it takes too long, and it lacks agility to react to changes during and after acquisition.

Traditional testing techniques, such as scenario-based testing, are commonly used for assessing dependability of Navy systems. These techniques are strongly dependent on a particular system configuration and environment. The environment is usually modeled using flat, uniform distributions of software inputs and a limited number of profiles. Accordingly, the environment's profile and the most relevant estimates of the application inputs are considered. For example, in Navy's control systems, input parameters such as the number of weapons or the number of strike elements are included within the testing profiles.

The drawback of these techniques is that when the system configuration or environment changes, the designed test cases also need to be changed. Plugging in a new component will lead to a completely different system and will likely invalidate the test scenarios and profiles previously used. A similar problem also occurs when the application has to be used in an operational environment other than the one for which it was originally designed, which is expected to be common for reusable components. This raises an

important concern since Navy systems are submitted to frequent changes. Better ways of doing testing and evaluation are, thus, highly desirable.

Acquisition of new system modules and components is also an important concern. While an architectural or modular approach should allow for a certain degree of predictability, current Navy testing processes do not deal with modularity. As a result, time-consuming and expensive test procedures are needed each time a new system release comes up because available testing methods cannot support the high frequency of releases. Methods that limit the number of configurations of an architecture might be required, at least in the near term. Such limitations may be able to be relaxed as technologies for testing families of systems improve.

In flexible, open systems, components need to be assembled in a large number of configurations; and because the system is open, new components can be added that did not even exist at the time the system was originally designed.

In practice, the number of possible configurations for an open system is very large, because each of many slots in an open architecture can be independently filled by several different specific subsystems. Because the number of choices for each slot must be multiplied together to produce the total number of possible configurations, the number of possibilities is astronomical for the kinds of systems designed by the Navy. For example, it has been estimated that avionics software systems have thousands of components and tens of thousands of connections. In principle, the number of configurations is unbounded because an unknown and unlimited number of new subsystems can be created in the future. One consequence of this is that it will be impossible to test all configurations, and that a majority of the possible configurations will not be tested at all. These ideas are graphically summarized in Figure 5.

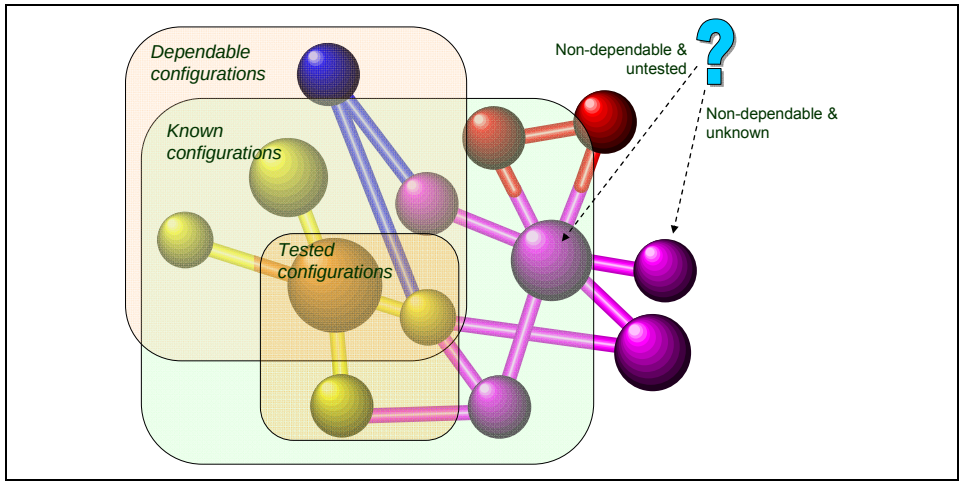

Figure 5. Example of Various Sets of Configuration Types: Dependable vs. Non-dependable, Known vs. Unknown, Tested vs. Untested

Each node in the figure represents a possible configuration of an open architecture. The connections between the nodes represent transitions between possible configurations, such as those resulting from the replacement of a subsystem with another plug-compatible subsystem that fits in the same slot of the open architecture. The figure is valid at many different scales; a module can be as small as a single data item, software procedure, or integrated circuit chip, and can include subsystems as large as entire ships. Figure 5 also highlights two important concerns (indicated by the interrogation point): the non-dependable

configurations that are unknown and the non-dependable configurations that (although known) have not been tested. Indeed, the number of configurations in practice is too large to be able to either know or test all of them. We seek alternative methods to palliate these issues.

These considerations indicate that quality assurance methods that depend on checking the individual possible behaviors of the entire system do not have any hope of being effective for the test and evaluation of flexible open architectures, and that conceptually new and different methods will have to be employed to achieve dependability for such systems in the presence of reuse and reconfiguration.

Testing of reusable subsystems is also subject to the above considerations and, similarly, requires new methods for effectively achieving dependability. This conclusion is consistent with past experience with system failures in military, scientific and commercial applications. The majority of observed failures are due to requirements and specifications errors, many of which manifest after a subsystem has been moved to a different environment than the one for which it was originally designed and tested. This is an indication that in current practice, the effectiveness of testing is very sensitive to the expected operating environment, which is unknown for reusable subsystems. Indeed, software reuse may invalidate the operational profiles and test cases and scenarios originally developed. The new operational profiles, test cases and scenarios are unknown, and no efficient method exists to calculate the required "delta" describing the necessary changes from previously used profiles (or test cases or scenarios) so that they can be applied to the newly reconfigured system. Open Architecture facilitates software reuse, which adds weight to this issue.

Test cases correspond to the traditional artifact used in testing, which are based on a model of the system environment. In stovepipe systems, requirements analysis and testing is greatly simplified compared to open systems. Also, there exist numerous methods and techniques that allow for linking the testing results to dependability parameters, so as to obtain a quantitative measure of the overall dependability of the system (e.g., notion of "dependability benchmarking").

The traditional concept of system design is not focused on architectural "bits." An architecture is related to a family of systems, while a design is traditionally associated with a single instance of a system. Also, an architecture involves more complexity than the traditional notion of system "configuration." This is due to the fact that the "context" is included in the architecture, which is usually unknown, not well understood or difficult to accurately take into account.

The type of dependability properties to be tested is also an important concern. Making Navy systems dependable will require considering a certain level of system performance and availability as part of the dependability concern. Indeed, architectural changes can considerably impact Key Performance Parameters (KPP), availability and other system requirements. Other concerns relate to how testing can be applied to Navy systems that are based on migrating services (e.g., reconfiguration of service-based architectures) and how system developers and testers can be involved in the acquisition process. At the moment, it is not possible to accurately know how much it may cost to move towards an open architecture paradigm.

Proposed Approach

In the short term, the problems outlined in the previous section are being addressed by attempting to predict future needs and by limiting the allowed configurations accordingly. This has the advantage of minimizing impact on current development processes and organizations, and the disadvantages that cost of testing is sill large and proportional to the number of reconfigurations, and that in cases in which predictions of future needs turn out to be incorrect, reconfigurations will need time for lengthy retesting, or new configurations will have to be fielded without assurance of dependability. However, in the Navy's Open Architecture vision, the "plug and fight" process is supposed to be inexpensive and agile.

The main objective of the OA approach is to get away from monolithic designs and architectures, and gain the ability to replace bits of systems. The goal is to facilitate DoD/Navy systems acquisition. This requires a shift from scenario-based testing to architecture-based testing. The constraints expressing the most important dependability properties should be part of the architecture. The architecture should, thus, include not only components and connections but also constraints. Note that there are different types of constraints—encompassing requirements, capabilities and standards (capabilities are similar to requirements). A dependable architecture should have requirements associated with it, which means that certain dependability guarantees should be already reflected in the architecture itself. Then, testing is not only to be applied to the system implementation, but also to the architectural model.

Thus, fully realizing the open architecture vision requires a new paradigm for test and evaluation. We propose such a paradigm here, based on the concepts of dependability contracts, interchangeable software parts, and computer-aided enforcement of dependability contracts.

Current approaches to system development and testing are more analogous to individual craftsmanship than they are to modern concepts of mass production and interchangeable parts. Craftsmen used to build things by individually tuning mating parts until they properly fit together. In such a context, designs could be relatively informal and relatively rough. In a mass production environment, parts are built to standards with precisely specified tolerances, and it is up to the designer to determine and verify the tolerances necessary to make the design work for any combination of parts that meet the specified tolerances. An example that illustrates this problem is the manufacturing of a rifle using a set of interchangeable parts. This is different from having parts that need to be crafted individually. It is necessary to evaluate how much variation is allowed to make different components and parts fit into the rifle. To do this, it is necessary to measure absolute sizes and construct the various parts of the rifle with certain tolerances. These modular approaches have been used in manufacturing for many years, but have never been successfully integrated into software engineering approaches. Another example consists of modern audio systems. There exist specific standards for audio systems specifying how things need to fit together in order for components from different vendors to work together effectively. Standards for audio system components can be relatively simple and generic only because the requirements for stereo systems are very simple. An audio system is not concerned about whether it is playing a song or the news. For systems whose behavior is sensitive to the meaning of the data, new types of standards will be needed to accomplish a similar function. These examples raise questions related to the kinds of standards that need be considered to make system components interchangeable and how such changes may influence testing. The answers to these questions should take into account the fact that we

aim at testing pieces of the architecture versus standards, not versus (other pieces of) the system.

We are seeking analogous quality-assurance techniques for systems involving software. The fundamental operation of such an approach can be outlined as follows:

1. System-wide capabilities are characterized by a set of dependability properties that must hold in all acceptable system configurations. These properties comprise the dependability contract for the system as a whole. They become part of a dependable open architecture for the system and serve as the basis for system quality assurance. Dependability contracts are primarily technical rather than legal documents, and they are intended to be checkable via software.

2. The designers of the open architecture determine the common structure of the system and develop the component-level dependability contracts for the subsystems and connectors. The common structure consists of connection patterns and subsystem slots to which all configurations must conform.

3. The quality-assurance team checks the structure of the architecture and the dependability contracts for subsystems and connectors to make sure they are strong enough to guarantee the system-wide dependability properties in all possible configurations. This is a one-time process that uses symbolic analysis techniques. Assuring the feasibility of this step is one of the objectives of ongoing research by the authors.

4. The quality assurance team tests each component (subsystem and connector) against its dependability contract. This is envisioned to be an automated process to enable sufficient large sets of test cases for statistically significant conclusions about desirable dependability levels. The cost for this step is proportional to the number of components, and the process must be completed once for each version of each atomic component. Technologies for doing this are well known, and many of them are used in common practice.

5. The quality-assurance team checks components for non-interference. This process is computer-aided. Many of the technologies for this are well known, and some of them are commonly used. Some development may be needed to get a complete set. This part of the process ensures that components that work correctly in isolation will continue to do so when they are connected.

6. The assumptions about the operating environment on which the architecture depends are checked by runtime monitoring. This can be done using BIT (Built-In-Test) technology that is currently in use in some DoD systems. This is recommended for all reusable components.

Figure 6 provides an overview of the global approach. The architectural and testing visions of the proposed approach are described below:

- Architectural vision
    - Consider an architecture as a support system not only for development but also for testing—including interchangeable software parts.

- Look at an architecture as consisting not only of components, connections and constraints, but also of standards, requirements/capabilities and environmental assumptions.

- Testing vision
  - Relate testing to standards and constraints as a means to ensure architecture meets requirements and provides the needed capabilities.
  - Relate standards to architectural structures and associated dependability requirements.
  - Certify absence of interference between components and the dependability properties of interest.
  - Check constraints on environment at reconfiguration time.
  - The purpose is to prevent problems (such as integration problems). When feasible, this is better than detecting those problems. The approach should allow for making responsibilities more visible.

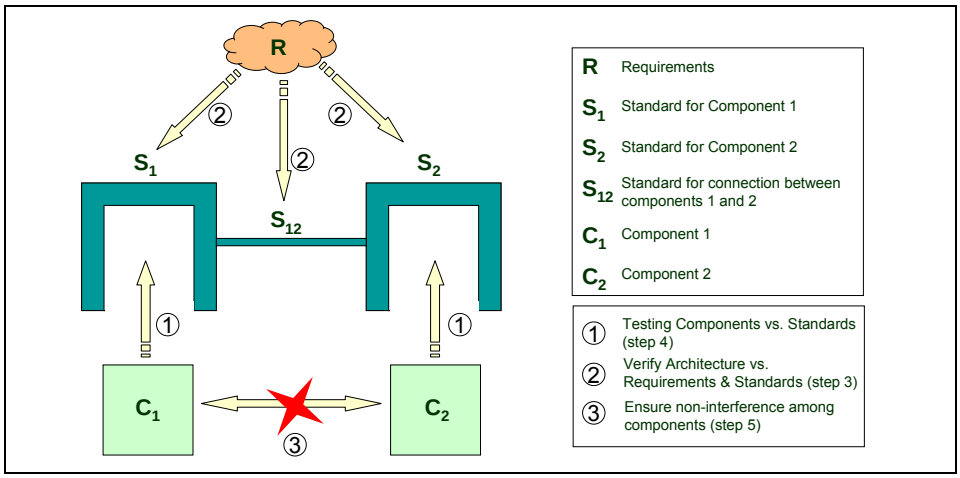

Figure 6. Overview of the Global Approach

The above process is a long-term goal whose realization depends on refinement and integration of new technologies and processes—especially those supporting steps 3 and 5.

Full success of the approach will eliminate the need for integration testing after each reconfiguration. This is the meaning of interchangeable software components. We do not propose to eliminate integration testing entirely, even in the long term. The reason is that all analysis is relative to a model. While the models we use are good, it is always possible that the existing implementation does not realize the intended model completely precisely. For example, it is possible that the compiler used does not implement its programming language correctly in some rare cases, or that the hardware does not perform its functions as specified under some rare conditions. For these reasons, we recommend integration testing for at least one system configuration, e.g., the initial configuration to be fielded. Shorter-term reductions in the amount of testing needed after a reconfiguration are expected when effective non-interference checks eliminate specific kinds of failures due to integration issues.

Some examples include interference due to data or control interactions that are not allowed by the architecture, or due to resource constraints such as limits on memory, network bandwidth, or computation time. An example of existing technology that can eliminate a specific type of interference is architecture-based schedulability analysis, which can guarantee absence of failures due to real-time constraints and computing resource limits.

Another related issue is how to certify a standard. The certification method should be able to satisfy the critical requirements of all architectural configurations. Quality assurance and analysis techniques (such as model checking or theorem proving) could be used for such purposes. These techniques are well-known, well-established and have been used for many years. However, these techniques do not scale-up well. The reason is that they have traditionally been applied to program code, which is a very large artifact. We believe this technology can be applied to the architecture of a system because the architecture is much smaller than the code. This is especially the case if each level of the architectural hierarchy can be checked separately. To make this possible, the traditional concept of architecture should be enhanced, e.g., constraints and standards should also be included.

Another issue is that to check the absence of interference between components, static-analysis techniques (e.g., type checking, static checking, code analysis) will be needed, since testing is not enough for this purpose. This means that reachability analysis techniques will necessarily be different. The underlying approach might be “large scale.” but it does not mean it needs to be sophisticated (just feasible). Moreover, if testing is conducted against a standard, it is possible to have an automated testing oracle. In classical reliability techniques, it is possible to calculate the number of test cases needed to assure (with a certain confidence level) that the system will not experience more than a given number of failures during a determined period of time. For example, if the system should not fail more than once in N executions, the number of test cases needed for a confidence level of 1-1/N is given by Nlog2 N. (e.g., about 20 million test cases are needed to reach 10EXP-6 assurance).

Testing with respect to standards can drastically reduce the number of test cases needed because each component can be tested separately, and all possible combinations do not need to be checked. However, this source of potential savings depends on effective methods for carrying out steps 3 and 5 above.

Some shorter-term savings can be achieved by using testing approaches that obtain information about many different configurations based on a single test case run on a single configuration. An example is an approach that tests every pair of components that are connected in the architecture in at least one system configuration, but not in all possible contexts.

The major contributions and advantages of the proposed approach are:

- *Ability to reduce the testing effort*. The approach will enable reducing unnecessary testing on every system change and enable identifying what kinds of testing and checking do need to be repeated when something changes.
- *Ability to limit the retesting scope*. The approach will limit the scope of retesting when possible. This will involve a combination of testing with other kind of quality-assurance techniques.

- *Ability to assure dependability*. The approach will include methods for assuring, with a single analysis, that all possible configurations that can be generated in a model-driven architecture will satisfy given dependability requirements. Prior successful experience with developing methods of this kind has been demonstrated by the first author of this paper (Berzins, 2000). These results will be extended and applied to fit the requirements of the Navy open-architecture initiative.

Example

A simple example of part of a dependable architecture is shown in Figure 7. There are two component slots, one representing the software driver for a position sensor, and the other a control software module for an autopilot. There is one connector that carries information about the current position of the host platform. The example shows just a fragment of a realistic architecture as indicated by the ellipsis on the connections. In a complete architecture, the position information will also feed into other systems, such as tactical displays and weapons control systems.

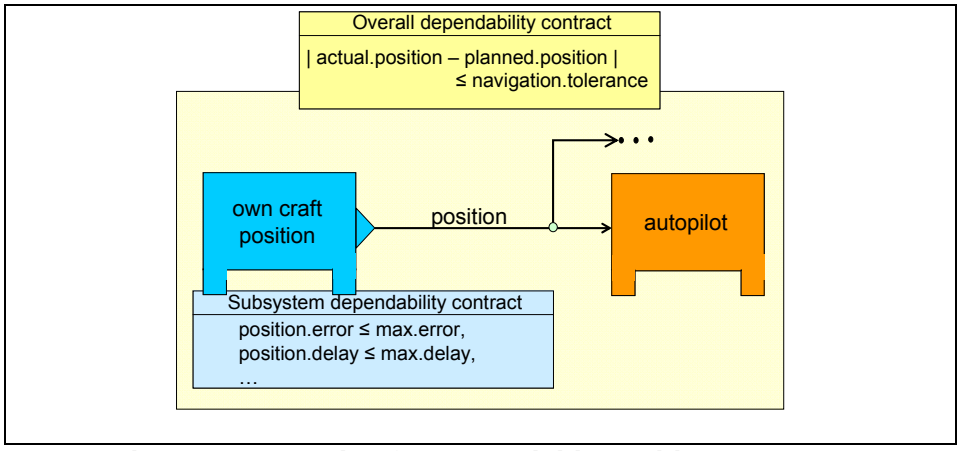

Figure 7. Example of a Dependable Architecture Fragment

The figure shows a simplified[1] partial description of the dependability contracts. The overall purpose of the interconnection is to keep the platform on course. This informal intent is expressed as a measurable dependability property that becomes the basis for the quality assurance of this architectural fragment. The navigation tolerance is a parameter of the overall system requirements as well as of the architecture. It provides a partial characterization of mission needs and a family of system configurations that meet that need. For example, different types of platforms may have different navigation tolerances. Note that this same architecture fragment is relevant to surface, subsurface and air platforms. The own-craft-position subsystem slot can be filled by a variety of sensors, such as GPS, inertial, VOR/DME, etc., and the autopilot subsystem slot can likewise be filled with components that realize different control algorithms. The subsystem dependability contract expresses part of the standards that any acceptable realization of the subsystem must meet—by expressing tolerances—for the accuracy of the sensor and the allowable time delay between the time the platform's position is measured and the time the position is delivered to the connector. It is the responsibility of the designers of the architecture to determine how the values of these

[1] For example, for air platforms, the vertical navigation tolerance can be different than the horizontal tolerance.

subsystem dependability parameters are derived from the overall dependability parameter. The purpose of Step 3 in the proposed quality assurance process is to check that this derivation is valid in the sense that the system will meet its requirements for any choice of sensors that meets its dependability contract, as well as for any choice of control algorithm that meets its dependability contract (not shown). This process depends on mathematical modeling, analysis and proof techniques, some existing and some to be developed.

The process of reconfiguring the system fragment in the example would amount to replacing the sensor and its software driver with another one. The quality-assurance activities associated with this would be certifying that the new component meets the dependability properties in the own-craft-position dependability contract (Step 4) and non-interference checks between the new component and the other components in the new configuration (Step 5).

Our objective is to provide static analysis methods to accomplish Step 5. If a complete set of such methods can be provided, then integration testing will not be needed after such a component replacement. If some but not all of the potential interference modes can be ruled out by static checking, then some integration testing will still have to be performed as part of Step 5, but the scope of that testing can be focused on the failure modes that are not yet covered by static checks. We note that although we have been mostly focused on replacement of software components, sometimes, as in this example, a meaningful reconfiguration may involve replacement of some hardware as well. In our example, some kinds of improvements may be possible by replacing just the driver software for a given sensor, but the largest gains may come from combining a new and more accurate type of position sensor with the new software driver needed to make the new sensor fit the existing subsystem slot in the architecture. The goal is not to change the architecture when the system is reconfigured. In such cases, the non-interference tests may include electrical, thermal and mechanical considerations in addition to software consideration.

Our recommendation is to identify potential sources of interference in detail, and to develop specific quality-assurance techniques for assuring absence of each type. These can involve a combination of static analysis checks, such as: data-type consistency, lack of unspecified data flow, lack of unspecified control flow, conformance to power and heat load limits, etc., with conventional testing processes. We also note that in some specific contexts, specialized efficient testing procedures are possible, for example, where dominance relations exist. For instance, in continuous domains it is common that a single worst-case test case can expose all the faults that any other test case could detect.

The dependability contracts in the example also have a dominance property: if a component has been certified with respect to a component with a larger error tolerance, it will also work for one with a smaller error tolerance, because every possible behavior of the more accurate component is also a possible behavior of the original, less accurate component. In the example, a sensor with a given max.error and max.delay can be replaced with any other sensor that has a smaller max.error and smaller max.delay provided that the new sensor also passes all non-interference checks.

We note that a kit of available components can be pre-certified with respect to Steps 4 and 5. This would enable agile dependable reconfiguration, and perhaps even a capability for on-the-fly “plug and fight.” The cost to do Step 4 is proportional to the number of components, and can economically be completed in advance. The cost to do Step 5

depends on whether generic non-interference methods can be developed for all needed failure modes. In the best case, it is proportional to the number of components and could be done in advance. If all-pairs analysis is necessary, cost would be quadratic in the number of components—making pre-checking expensive but still perhaps feasible in advance if the number of components in the reconfigurable part is not too big. In the worst case, where multiple interactions may be significant, some non-interference checking may still be required after reconfiguration, when the actual set of components in the new configuration is known.

Comparison with Related Work

The purpose of this section is to provide a comprehensive survey on existing approaches for improving Quality Assurance properties of open and flexible architecture-based systems. It is also an objective to review existing works on how testing is performed in a fluid environment with agile reconfiguration.

Comparison with Navy's Approaches

The Naval OA program interacts with the OACE, FORCEnet and MOSA initiatives in different ways. As described above, OACE is based on a set of standards for the computing environment of surface ship-centric systems specifications; MOSA is an acquisition and design approach, while FORCEnet is a unifying concept for multiple architectures and standards efforts in the Navy. The recommended testing practices are described by these standards in general terms and are mostly founded on scenario-based techniques. For example, OACE recommends functional and performance testing against specified system requirements, organized according to test cases and scenarios. It defines the concept of "virtual homogeneity" to facilitate testing by identifying groups of sub-systems performing similarly. The concepts of "tree of subsystems" and "aggregations of components" are also introduced. Each aggregation exists only in a manageable number of configurations. A test case can be applied to many configurations when there is no (considerable) interaction between choices of configurations. Schedulability analysis is recommended for ensuring that any configuration that the resource manager creates is schedulable. These are existing attempts to reduce cost of testing by limiting flexibility of systems and to increase confidence that a test case provides useful information about more than one configuration by limiting possible sources of interference between components.

Our methodology aims at defining a broader testing approach covering both functional and non-functional properties of Open Architecture-based systems, with emphasis on ensuring dependability for all possible system configurations. Instead of seeking for subsystems performing similarly (concept of "virtual homogeneity"), our approach will use architectural artifacts and standards which already define the basis for all the different groups of subsystems that can be developed in practice. In our context, "performing similarly" means "meeting the dependability contract associated with a subsystem slot in the open architecture." Since our approach will work at the architectural level, and the architecture represents a family of systems and subsystems, the concepts of "tree of subsystems" and "aggregation of components" will be also covered. The non-interaction between choices of configurations is already covered by the concept of non-interference defined in our approach. Schedulability analysis is also part of the non-interference notion, since it will allow for predicting resource conflicts between tasks and processes.

Comparison with Component-based Testing

Component-based testing can be readily employed for Step 4 of our methodology to further test a candidate software component against the specific domain and architectural standards of the target system in which it is to be plugged-in and integrated.

Traditionally, component-based testing is performed by the component's developer itself (e.g., through unit testing). It is aimed at establishing the proper functioning of the component and at detecting possible failures early, i.e., ensuring the quality of the component before it is released. The tests established by the developer can rely not only on a complete documentation and knowledge of the component, but also on the availability of the source code, and, thus, in general pursue some kind of coverage testing. Therefore, when applied by the component's developer, this testing approach cannot address the verification of the component's behavior with respect to the specifications of the host system(s) in which the component will be later assembled (i.e., integration and system testing). Note, however, that component-based testing techniques are also used by system testers and integrators.

Voas (1998, June; 2000, August) proposed a certification strategy for off-the-shelf components relying on black-box testing, system-level fault injection and defense protection through wrapping. Black-box testing is a well-known testing technique used whenever the source code of a component is not available, only its interface specifications. System fault injection and defense wrapping are system-level approaches for integration testing and fault containment that might not be needed in our approach if the non-interference property is fully satisfied.

Other approaches aim at making component's data available (e.g., internal behavioral and structural data, development data, etc.) so that the data can assist the testing process. The work in Orso, Harrold and Rosenblum (2001) defines an approach in which metadata of a component (describing both static and dynamic aspects) are available throughout the entire component's lifecycle. The feasibility of the approach is demonstrated in the context of component-based testing, consisting of the generation of self-checking code and program slicing. The work in Whaley, Martin and Lam (2002) automatically extracts a finite-state machine model from the interface of a software component, which can be delivered along with the component itself for testing purposes. Off-the-shelf (OTS) components are usually acquired as black-box code without access to data that might be necessary for testing. Salles, Rodriguez, Fabre and Arlat (1999) developed a framework for integration testing of OTS real-time operating systems (RTOS). Information needed for testing is obtained through reflective techniques implemented in an additional software module added to the OTS component. A fault-injection methodology is used to verify that the behavior of the integrated OTS component does not impact the dependability of the system.

Bertolino and Polini (2003) recognized the importance of testing a software component in its deployment environment (i.e., the target system). They developed a framework that supports functional testing of a software component with respect to customer's specification—which also provides a simple way to enclose the developer's test suites which can be re-executed by the customer. The customer is thus provided with both a technique to specify a deployment test suite early and an environment for running and reusing the specified tests on any component implementation. There is a complete decoupling between the tests' specification and component implementation. The approach

requires the customer to have a complete specification of the component to be incorporated into a system.

In the formal verification domain, there has been a long history of research on verification of systems with modular structure. A key idea (Lamport, 1983; Kupferman & Vardi, 1997; Henzinger, Qadeer & Rajamani, 1998) in modular verification is the assume-guarantee paradigm: a module should guarantee to have the desired behavior once the environment with which the module is interacting has the assumed behavior. There have been a variety of implementations for this idea (see, e.g., Grumberg & Long, 1994; Alur et al., 1998; Pasareanu, Dwyer & Huth, 1999; Dingel, 2003; Chaki, Clarke, Groce, Jha & Veith, 2003; Xie & Browne, 2003). The key issue with the assume-guarantee style reasoning is how to obtain assumptions about the environment. Giannakopoulou et al. (Giannakopoulou, Pasareanu & Barringer, 2002; Giannakopoulou, Pasareanu & Cobleigh, 2004) introduced a novel approach to generate assumptions that characterize exactly the environment in which a component satisfies its property. Their idea is based on a purely formal verification technique (model-checking). Fisler et al. (Fisler & Krishnamurthi, 2001; Li, Krishnamurthi & Fisler, 2002) introduced a similar idea of deducing a model-checking condition for extension features from the base feature for model-checking, feature-oriented software designs. This approach is not applicable to component-based systems where unspecified components exist. This work differs from related work like Xie and Dang (2004), in which an automata-theoretic approach is used to solve a similar LTL model-checking problem.

In the past decade, there has also been significant research on combining model-checking and testing techniques for system verification, which can be grouped into a broader class of techniques called specification-based testing. Many of the studies utilize model-checkers' ability of generating counter-examples from a system's specification to produce test cases against an implementation (Callahan, Schneider & Easterbrook, 1996; Holzmann, 1997, May; Engels, Feijs & Mauw, 1997; Gargantini & Heitmeyer, 1999; Ammann, Black, & Majurski, 1998; Black, Okun, & Yesha, 2000). Peled et. al. (Peled, Vardi & Yannakakis 1999: Groce, Peled & Yannakakis, 2002; Peled, 2003) studied the issue of checking a black-box against a temporal property (called black-box checking). The research focuses on how to efficiently establish abstract models for black-box testing and on how to define properties (e.g., LTL formula) about the black-box components.

Comparison with Runtime Software Reconfiguration

For an important class of safety- and mission-critical software systems, such as air traffic control, telephone switching, and high-availability public information systems, shutting down and restarting the system for upgrades incurs unacceptable delays, increased cost, and risk. Support for runtime modification is a key aspect of these systems. In our methodology, a reconfigured set of components can be seen as a particular configuration of a system architecture. Since our approach aims at guaranteeing dependability properties for the family of systems and configurations represented by the architecture, the proposed testing approach should be able to provide assurance in presence of runtime reconfiguration for at least a certain number of properties (e.g., non-interference).

There are a wide variety of techniques for supporting runtime software change. Some of the most popular techniques are based on Dynamic Software Architectures (Oreizy, 2007). Several research projects have addressed these issues, such as Self-Adaptive, Healing Architectures (ArchShell, 2007), or Dynamic Wright (Allen, Douence & Garlan, 1998, April). Gupta, Jalote and Barua (1996, February) describe an approach to

modeling changes at the statement and procedure levels for a simple imperative programming language. Many dynamic programming languages, such as Lisp, Smalltalk, and Haskel (Peterson, Hudak & Ling, 1997, July) have supported runtime software change for decades. Dynamic linking mechanisms and libraries have been available in operating systems such as UNIX, Microsoft Windows, and the Apple Macintosh for some time. New approaches to dynamic linking (Franz, 1997, March) hope to significantly reduce the runtime performance overhead associated with using such mechanisms. Dynamic Object Technology, such as CORBA (Object Management Group, 1996, July) and COM (Brockschmidt, 1994) support the runtime locations, loading, and binding of software objects or components.

Service-oriented architectures (SOAs) typically have a dynamic nature, given by the runtime detection of components through registry services and subsequent dynamic binding. The work in Baresi, Heckel, Thone, and Varro (2006) defines a refinement relation from a generic style of component-based systems to the SOA style based on the use of graph transformation systems as models of architectural styles at different levels of platform abstraction (which represent reconfiguration and communication scenarios as graph transformation sequences). Besides the many proposals for Architecture Description Languages (ADLs), like Rapide (Luckham et al., 1995; Oreizy, 1996, August; Oreizy, Medvidovic & Taylor, 1998, April), Wright (Allen, 1997; Allen, Douence & Garlan, 1997, September; Allen, Douence & Garlan, 1998, April; Allen, Douence & Garlan, 1998), Darwin (Magee, Dulay, Eisenbach & Kramer, 1995; Kramer & Magee, 1998) or C2 (Medvidovic, 1996, October; Oreizy, Medvidovic & Taylor, 1998, April), we must mention those approaches that exploit graph transformation (Hirsh, 2003; Hirsh & Montanari, 2001, August; Metayer, 1996, October; Taentzer, Goedicke & Meyer, 2000; Wermelinger & Fiadeiro, 2002; Baresi, Heckel, Thone, & Varro, 2003; Gonczy, 2006) to reason about the consistency of reconfiguration operations and interaction of components with respect to structural constraints. Le Metayer (1996, October) describes architectures by graphs and the valid graphs of an architectural style by a graph grammar. Reconfiguration is described by conditional graph-rewriting rules. He uses static-type checking to prove that the rewriting rules are consistent with the respective style. The graphs represent computational entities but not connectors, specifications, or other resources. Wermelinger and Fiadeiro (2002) provide an algebraic framework based on Category theory in which architectures are represented as graphs of CommUnity programs and superpositions. Dynamic reconfigurations are specified by graph transformation rules over architecture instances. Both styles and rules are used for modeling domain-specific restrictions rather than the underlying platform. Consequently, they do not deal with refinement relationships between different levels of platform abstraction. Hirsch (2003) uses hypergraphs to represent architectures and hyperedge replacement grammars to define the valid architectures of an architectural style. Furthermore, he uses graph transformation rules to specify runtime interactions among components, reconfigurations, and mobility. In the CHAM approach (Inverardi & Wolf, 1995, April), architectural reconfiguration is studied in terms of molecules and reactions, and the proposals that represent architectural styles by means of graph grammars (Hirsh & Montanari, 2001, August; Metayer, 1996, October; Taentzer, Goedicke & Meyer, 2000; Wermelinger & Fiadeiro, 2000, March) and reason on changes and evolution with respect to structural constraints. Some of these approaches use a graph grammar to specify the class of admissible configurations of the style. Graph transformation rules model only dynamic aspects like evolution and reconfiguration. The advantage is that a declarative specification is more abstract and easier to understand, even if constructive/operational ones are better for analysis and tools. The use of graph-transformation techniques to capture dynamic semantics of models has also been inspired by work proposed by Engels,

Hausmann, Heckel and St. Sauer (2000) under the name of dynamic meta-modeling. That approach extends metamodels—re-defining the abstract syntax of a modeling language like UML by using graph-transformation rules that allow for describing changes to object graphs and represent the states of a model.

Grammar-oriented Programming (GOP) and Grammar-oriented Object Design (GOOD) (GOOD, 2002) are based on designing and creating a domain-specific programming language (DSL) for a specific business domain. GOOD can be used to drive the execution of the application, or it can be used to embed the declarative processing logic of a context-aware component (CAC) or context-aware service (CAS) (Arsanjani, Curbera, & Mukhi, 2004). GOOD is a method for creating and maintaining dynamically reconfigurable software architectures driven by business-process architectures. The business compiler was used to capture business processes within real-time workshops for various lines of business and create an executable simulation of the processes used. Instead of using one DSL for the entire programming activity, GOOD suggests the combination of defining domain-specific behavioral semantics in conjunction with the use of more traditional, general purpose programming languages.

The use of model-checking techniques for verifying software architectures has been thoroughly studied. For example, vUML (Lilius & Paltor, 1999, October), veriUML (Compton, Gurevich, Huggins & Shen, 2000), JACK (Gnesi, Latella & Massink, 1999), and HUGO (Schafer, Knapp & Merz, 2001) support the validation of distributed systems (where each statechart describes a component), but do not support complex communication paradigms. These works study static systems whose topology cannot vary at runtime. Similarly, Garlan Khersonsky and Kim (2003, May) and the researchers involved in the Cadena project (Hatcliff, Deng, Dwyer, Jung & Ranganath, 2003, May) applied model-checking techniques to analyze specific architectures with a fixed topology based on the publish/subscribe paradigm. A formal approach that considers refinement of dynamic reconfiguration can be found in Bolusset and Oquendo (2002). The approach is targeted on the translation from one ADL to another rather than on the refinement between architectural styles. Cherchago and Heckel (2004) describe the application of graph transformations in the runtime matching of behavioral Web service specifications. In Heckel and Mariani (2005), the conformance testing of Web services is based on graph transformations, focusing on the automated test-case generation. The work of Bertolino and Polini (2006) utilizes the benefits of these approaches and defines fault-tolerant algorithms incorporated into appropriate reconfiguration mechanisms for modeling reliable message delivery by graph-transformation rules in SOA. Graph transformation is used as a specification technique for dynamic architectural reconfigurations in Wermelinger and Fiadeiro (2002), using the algebraic framework CommUnity. Hirsch uses graph transformations over hypergraphs (2003) to specify runtime interactions among components, reconfigurations, and mobility in a given architectural style. A profile for reliability was designed for J2EE applications in Rodrigues, Roberts, Emmerich and Skene (2004). In Zheng, Jun and Yan (2005), a pattern-based specification and runtime validation approach is presented for interaction properties of web services using a semantic web rule language (SWRL). GROOVE (Graphs for Object-oriented Verification, 2007) is a project centered around the use of simple graphs for modeling the design-time, compile-time, and runtime structure of object-oriented systems; it also focuses on graph transformations as a basis for model transformation and operational semantics. This entails a formal foundation for model transformation and dynamic semantics, and the ability to verify model transformation and dynamic semantics through an (automatic) analysis of the resulting graph transformation systems—for instance using model checking.

The techniques described above such as model checking and graph transformations can benefit several steps of our methodology. In Step 2 these techniques can be used to derive dependability contracts addressing reconfiguration, topology and connections properties and constraints at the system and subsystem levels. In Steps 3 and 4, these techniques can be used in combination with other symbolic analysis techniques and testing techniques for verifying the structure of the architecture and the dependability contracts. These techniques can also provide useful information about the various sources of interference between the target components and the host system, and can help determine suitable and alternate approaches to avoid those interference sources.

However, designers have traditionally sought alternatives to runtime change, especially for safety-critical applications such as combat systems. Several reasons account for this:

1. *It is usually avoidable*. Runtime change is not a critical aspect of many software systems, and several techniques have been devised to circumvent the need for runtime change altogether. Regularly scheduled downtimes, functional redundancy or clustering, and manual overrides are all examples of such techniques.
2. *It increases risk*. System integrity, reliability, and robustness are more difficult to ensure in light of runtime change.
3. *It increases cost*. There is typically a marked performance overhead associated with supporting runtime change. Additionally, few techniques have limited expertise; and a lack of proven techniques for supporting runtime change exasperates engineering costs.

Although “plug and fight” has been articulated as a goal, in the near term reconfiguration is likely to be more constrained and less agile due to weapons certification and doctrine issues.

Conclusion

This paper explores methods for test and evaluation of flexible systems with open architectures, and proposes an approach for substantially reducing the amount of testing necessary for dependable reconfigurable systems. The approach involves augmenting open architectures with measurable dependability properties associated with the system as a whole as well as dependability properties associated with slots for replaceable subsystems. It also involves augmenting testing with other kinds of quality-assurance methods. These additional methods include static checks for non-interference properties. The purpose of these checks is to ensure that components that work correctly in isolation will continue to do so in the context of a given dependable open architecture. In the long term, this approach should eliminate the need for integration testing after each reconfiguration, and in the short to medium term, it should substantially reduce the amount of integration testing required after reconfiguration.

References

Allen, R. (1997). A formal approach to software architecture (PhD thesis). School of Computer Science, Carnegie Mellon University.

Allen, R., Douence, R. & Garlan, D. (1997, September). Specifying dynamism in software architectures. In *Proceedings of the workshop on foundations of component-based systems* (pp.11-22). Zurich,Switzerland.

Allen, R., Douence, R. & Garlan, D. (1998, April). Specifying and analyzing dynamic software architectures. In *Proceedings of the conference on fundamental approaches to software engineering. Lecture Notes in Computer Science*, 1382.

Allen, R., Douence, R., & Garlan, D. (1998). Specifying and analyzing dynamic software architectures. In *Proceedings of 1998 conference on fundamental approaches to software engineering.* Lisbon, Portugal. Retrieved April 9, 2007, from http://www.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/wright-fase98.html

Alur, R., Henzinger, T.A., Mang, F. Y. C., Qadeer, S. Rajamani, S. K., & Tasiran, S. (1998). MOCHA: Modularity in model checking. In *Proceedings of the 10th international conference on computer aided verification* (pp. 521–525). *Lecture Notes in Computer Science*, 1427. New York: Springer.

Ammann, P., Black, P.E., & Majurski, W. (1998, December). Using model checking to generate tests from specifications. In *Proceedings of the 2nd IEEE international conference on formal engineering methods* (pp. 46-54). Brisbane, Qld., Australia. Washington, DC: IEEE Computer Society.

ArchShell. (2007). *Self-adaptive, healing architectures*. Retrieved April 9, 2007 from http://www.isr.uci.edu/architecture/dynamic-arch.html

Arsanjani, A., Curbera, F., & Mukhi, N. (2004). Manners externalize semantics for on-demand composition of context-aware services. In *Proceedings of IEEE international conference on web services* (pp. 583 – 590).

Baresi, L., Heckel, R., Thone, S., Varro, D. (2003). Modeling and validation of service-oriented architectures: application vs. style. In *Proceedings of ESEC / SIGSOFT FSE* (pp. 68-77).

Baresi, L., Heckel, R., Thone, S., Varro, D. (2006, June). Style-based modeling and refinement of service-oriented architectures—A graph transformation-based approach. *Software and Systems Modeling,* 5(2). New York: Springer-Verlag.

Black, P. E., Okun, V., & Yesha, Y. (2000, September). Mutation operators for specifications.. In *Proceedings of the15th automated software engineering conference (ASE2000)* (pp. 81-88). Grenable, France. Washington, DC: IEEE Computer Society

Bertolino, A. & Polini, A. (2003). A framework for component deployment testing. *ICSE*, pp. 221–231. Washington, DC: IEEE Computer Society. Retrieved April 9, 2007, from http://www1.isti.cnr.it/~polini/downloads/icse03/icse2003.pdf.

Bertolino, A., & Polini, A. (2006). Modeling of reliable messaging in service oriented architectures. In *Proceedings of international workshop on web services modeling and testing*. Palermo, Italy: WS-MaTe.

Berzins, V. (2000). Static analysis for program generation templates. In *Proceedings of the 2000 ARO/ONR/NSF/DARPA* workshop on modeling software system structures in a

fastly moving scenario. Santa Margherita Ligure, Italy. Retrieved April 9, 2007, from http://www.disi.unige.it/person/ReggioG/PROCEEDINGS/berzins.pdf.

Bolusset, T., & Oquendo, F. (2002). Formal refinement of software architectures based on rewriting logic. In *Proceedings, RCS 02 international workshop on refinement of critical systems*. Retrieved April 9, 2007, from www-lsr.imag.fr/zb2002/.

Brockschmidt, K. (1995, May). *Inside OLE 2.2nd Bk & Cdr edition.* Microsoft Press.

Callahan, J., Schneider, F., & Easterbrook, S. (1996). Automated software testing using model checking. In *Proceedings of 1996 SPIN workshop*.

Chaki, S., Clarke, E., Groce, A., Jha, S. & Veith, H. (2003). Modular verification of software components in C. In *Proceedings of ICSE'03* (pp. 385–395). Washington, DC: IEEE Computer Society Press.

Cherchago, A., & Heckel, R. (2004). Specification matching of web services Using Conditional Graph Transformation Rules. In *Proceedings of international conference on graph transformations*. *Lecture Notes of Computer Science*, 3256, 304-318. New York: Springer.

Chuang, S.-N., Chan, A.T.S., Cao, J., & Cheung, R. (2003, May). Dynamic service reconfiguration for wireless web access. In *Proceedings of 12th international world wide web conference* (WWW 2003). Budapest, Hungary.

Clarke, M., Blair, G., Coulson, G., & Parlavantzes, N. (2001, November). An efficient component model for the construction of adaptive middleware. In *Proceedings of the IFIP/ACM international conference on distributed systems platforms and open distributed processing (middleware)*. Heidelberg, Germany. New York: ACM.

Compton, K., Gurevich, Y., Huggins, J., & Shen, W. (2000). *An automatic verification tool for UML* (Technical Report CSE-TR-423-00). Ann Arbor, MI: University of Michigan, EECS Department.

DAU (2007a). *Naval open architecture—Terms & definitions*. Defense Acquisition University, Acquisition Community Connection. Retrieved April 9, 2007, from https://acc.dau.mil/CommunityBrowser.aspx?id=22108

DAU. (2007b). *Naval open architecture*. Module CLE012. DAU Virtual Campus Defense Acquisition University. Retrieved April 9, 2007, from https://learn.dau.mil/html/clc/Clc.jsp

DAU. (2007c). *Naval open architecture—Introduction to open architecture*. Module CLE012—Intro to OA. DAU Virtual Campus. Defense Acquisition University. Retrieved April 9, 2007, from https://learn.dau.mil/html/clc/Clc.jsp.

Deering, V., Grates, P., Hedge, T., Kung, S., Martinez, M., Mcarthy, P., Pugh, K., Radojkovic, S. (2006, September). *Open architecture as an enabler for FORCEnet* (NPS Technical Report # NPS-SE-06-002). Monterey, CA: Naval Postgraduate School.

Dingel, J. (2003). Computer-assisted assume/guarantee reasoning with verisoft. In *Proceedings of ICSE*'03 (pp. 138–148). Washington, DC: IEEE Computer Society Press.

DoD. (2003, May 12). *The defense acquisition system* (DoDD). Retrieved April 9, 2007, from http://www.at.hpc.mil/Docs/d50001p.pdf.

DoD. (2004, April 5). *Amplifying DoDD 5000.1 guidance regarding modular open systems approach (MOSA) implementation*. Washington, DC: Under Secretary of Defense. Retrieved April 9, 2007, from http://www.acq.osd.mil/osjtf/pdf/wynn_memo_04.pdf.

DoN. (2004, August 5). *Naval open architecture scope and responsibilities.* Assistant Secretary of the Navy (Research, Development & Acquisition). Retrieved April 9, 2007, from http://acquisition.navy.mil/content/view/full/4495.

Engels, A., Feijs, L., & Mauw, S. (1997). Test generation for intelligent networks using model checking. In *Proceedings of TACAS'97 (*pp. 384-398). *Lecture Notes in Computer Science*, 1217. New York: Springer.

Engels, G., Hausmann, J.H., Heckel, R., & St. Sauer. (2000). Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proceedings, UML 2000—The unified modeling language (323-337). Lecture Notes in Computer Science*, 1939. New York: Springer.

Fisler, K., & Krishnamurthi, S. (2001). Modular verification of collaboration-based software designs. In *FSE'01* (pp. 152–163). New York: ACM Press.

Flowers, K., & Azani, C., (2004). *Open systems policies and enforcement challenges*. In *Proceedings of the national defense industry association systems engineering conference.* Dallas, TX. Retrieved April 9, 2007, from http://www.acq.osd.mil/osjtf/pdf/os_policy.pdf.

FORCEnet. (2007a). Naval Network Warfare Command. Retrieved April 9, 2007, from http://forcenet.navy.mil/

FORCEnet. (2007b). *System view*. Retrieved April 9, 2007, from http://forcenet.navy.mil/architecture/sv.htm

Franz, M. (1997, March). Dynamic linking of software components. *IEEE Computer*, 30(3), 74-81.

Gargantini, A., & Heitmeyer, C. (1999). Using model checking to generate tests from requirements specifications. In *Proceedings of ESEC/FSE'99* (pp. 146–163). *Lecture Notes in Computer Science,* 1687. New York: Springer.

Garlan, D., Khersonsky, S., & Kim, J.S. (2003, May). Model checking publish-subscribe systems. In *Proceedings of the 10th SPIN workshop*. *Lecture Notes in Computer Science*, 2648.

Giannakopoulou, D., Pasareanu, C.S., & Barringer, H. (2002). Assumption generation for software component verification. In *Proceedings of ASE'02* (pp. 3–13). IEEE Computer Society.

Giannakopoulou, D., Pasareanu, C.S., & Cobleigh, J.M. (2004). Assume-guarantee verification of source code with design-level assumptions. In *Proceedings of ICSE'04* (pp. 211–220). Washington, DC: IEEE Press.

Groce, A., Peled, D., & Yannakakis, M. (2002). Amc: An adaptive model checker. In *Proceedings of CAV'02* (pp. 521-525). *Lecture Notes in Computer Science*, 2404. New York: Springer.

Grumberg, O., & Long, D.E. (1994). Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16, 843–872.

GOOD. (2002). Grammar-oriented Object Design (GOOD) Homepage. Retrieved April 9, 2007, from http://www.arsanjani.org/GOOD/

Gonczy, L. (2006). Verification of reconfiguration mechanisms of service-oriented architectures. Retrieved April 9, 2007, from http://home.mit.bme.hu/~gonczy/publications/gonczy_cscs06.pdf

Gnesi, S., Latella, D., & Massink, M. (1999). Model checking UML statecharts diagrams using JACK. In *Proceedings of the 4th IEEE international symposium on high assuarance systems enginering (HASE)* (pp. 46–55).

Graphs for Object-Oriented Verification (GROOVE project). (2007). Retrieved April 9, 2007, from http://groove.sourceforge.net/groove-index.html

Gupta, D., Jalote, P., & Barua, G. (1996, February). A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2), 120-131.

Hatcliff, J., Deng, W., Dwyer, M.B., Jung, G., & Ranganath, V. (2003, May). Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the IEEE 25th international conference on software engineering* (pp. 160–172).

Heckel, R., & Mariani, L. (2005). Automated conformance testing of web services. In *Proceedings of 8th iternational conference on fundamental approaches to sofware engineering* (FASE 2005) (pp. 34-48). *Lecture Notes in Computer Science,* 3442. New York: Springer,.

Henzinger, T.A., Qadeer, S., & Rajamani, S.K. (1998). You assume, we guarantee: Methodology and case studies. In *Proceedings of CAV'98* (pp. 440–451). *Lecture Notes in Computer Science*, 1427. New York: Springer. Retrieved April 9, 2007, from http://mtc.epfl.ch/~tah/Publications/you_assume_we_guarantee.pdf.

Hirsh, D., & Montanari, M. (2001, August). Synchronized hyperedge replacement with name mobility. In *Proceedings, CONCUR 2001*, Aarhus, Denmark. *Lecture Notes in Computer Science*, 2154, 121–136. Aarcur, Denmark: Springer-Verlag.

Hirsh, D. (2003). *Graph transformation models for software architecture styles* (PhD thesis). Buenos Aires, Argentina: Departamento de Computacion, Universidad de Buenos Aires.

Holzmann, G.L. (1997, May). The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5—*Special Issue: Formal Methods in Software Practice)*, 279–295.

Inverardi, P., & Wolf, A. (1995, April). Formal specification and analysis of software architectures using the chemical abstract machine model*. IEEE Transactions on Software Engineering*, 21(4), 373–386. April 1996.

Kramer, J., & Magee, J. (1998). Analyzing dynamic change in software architectures: A case study. In *Proceedings of the fourth international conference on configurable distributed systems* (pp. 91–100).

Kupferman, O., & Vardi, M. (1997). Module checking revisited. In *Proceedings of CAV'97* (pp. 36–47). *Lecture Notes in Computer Science,* 1254. New York: Springer.

Lamport, L. (1983). Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2), 190–222.

Le Metayer, D. (1996, October 16-18). Software architecture styles as graph grammars. In *Proceedings of the fourth ACM SIGSOFT symposium on the foundations of software engineering. ACM Software Engineering Notes*, 216, 15–23. New York: ACM Press.

Li, H., Krishnamurthi, S., & Fisler, K. (2002). Verifying cross-cutting features as open systems. *ACM SIGSOFT Software Engineering Notes*, 27(6), 89–98.

Lilius, J., & Paltor, I.P. (1999, October). vUML: a tool for verifying UML models. In *Proceedings of the 14th IEEE international conference on automated software engineering (ASE)* (pp. 255–258).

Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D., & Mann, W. (1995). Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering,* 21(4), 336–355.

Magee, J., Dulay, N., Eisenbach, S., & Kramer, J. (1995). Specifying Distributed Software Architectures. In *Proceedings of ESEC 95—5th European software engineering conference* (pp. 137–153). *Lecture Notes in Computer Science,* 989. New York: Springer.

Medvidovic, N. (1996, October). ADLs and dynamic architecture changes. In A. L. Wolf, (ed.), *Proceedings of the second international software architecture workshop (ISAW-2)* (pp. 24-27). San Francisco, CA,.

Open Systems Joint Task Force. (2007). *MOSA defined.* Retrieved April 9, 2007, from http://www.acq.osd.mil/osjtf/mosadef.html.

Naval Sea Systems Command. (2007). *OACE*. Open Architecture Computer Environment (OACE), Naval Sea Systems Command, Naval Surface Warfare Center, Dahlgren Division/Laboratory. Retrieved April 9, 2007, from http://www.nswc.navy.mil/wwwDL/B/OACE/.

NSWCDD. (2004, August 23a). *Open architecture (OA) computing environment design guidance (*version 1.0). Naval Surface Warfare Center Dahlgren Division (NSWCDD). Retrieved April 9, 2007, from http://www.nswc.navy.mil/wwwDL/B/OACE/docs/OACE_Design_Guidance_v1dot0_final.pdf.

NSWCDD (2004, August 23b). *Open architecture (OA) computing environment technologies and standards* (version 1.0). Naval Surface Warfare Center Dahlgren Division (NSWCDD). Retrieved April 9, 2007, from http://www.nswc.navy.mil/wwwDL/B/OACE/docs/OACE_Tech_Stds_v1dot0_final.pdf.

Object Management Group. (1996, July). *The common object request broker: Architecture and specification* (Revision 2.0). OMG Technical Report.

Oreizy, P. (2007). *Dynamic software architectures resources*. Retrieved April 9, 2007, from http://www.ics.uci.edu/~peymano/dynamic-arch/

Oreizy, P. (1996, August). *Issues in the runtime modification of software architectures*. (Technical Report UCI-ICS-96-35). Irvine, CA: University of California, Irvine.

Oreizy, P., Medvidovic, N., & Taylor, R.N. (1998, April). Architecture-based runtime software evolution. In *Proceedings of the IEEE 20th international conference on software engineering* (pp. 177-186). Kyoto, Japan.

Orso, A., Harrold, M.J., & Rosenblum, D. (2001). Component metadata for software engineering tasks. *Lecture Notes in Computer Science*, 1999, 129–144. Retrieved April 9, 2007, from http://www.cc.gatech.edu/aristotle/Publications/Papers/edo00.pdf.

Pasareanu, C.S., Dwyer, M.B., & Huth, M. (1999). Assume-guarantee model checking of software: A comparative case study. *SPIN*, 168–183.

Peled, D. (2003). Model checking and testing combined. In *Proceedings of ICALP'03* (pp. 47–63). *Lecture Notes in Computer Science,* 2719. New York: Springer.

Peled, D., Vardi, M.Y., & Yannakakis, M. (1999). Black box checking. In *Proceedings of FORTE/PSTV'99* (pp. 225–240). New York: Kluwer.

Peterson, J., Hudak, P., & Ling, G.S. (1997, July). *Principled dynamic code improvement* (Yale University Research Report YALEU/DCS/RR-1135). New Haven, CT: Department of Computer Science, Yale University.

Rodrigues, G.N., Roberts, G., Emmerich, W., & Skene, J. (2004). Reliability support for the model driven architecture. In *Proceedings, workshop on software architectures for dependable systems* (WADS 2003) (pp. 79-98). *Lecture Notes in Computer Science*, 3069. New York: Springer.

Salles, F., Rodriguez, M., Fabre, J.-C., & Arlat, J. (1999). MetaKernels and fault dontainment wrappers. In *Proceedings of the 29th fault-tolerant computing symposium* (pp. 22-29). Retrieved April 9, 2007, from http://ieeexplore.ieee.org/iel5/6328/16917/00781030.pdf?tp=&isnumber=&arnumber=781030

Schafer, T., Knapp, A., & Merz, S. (2001). Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3), 1-13.

Shannon, J. (2006). Naval enterprise open architecture—Open architecture assessment tool training. Retrieved April 9, 2007, from https://acc.dau.mil/GetAttachment.aspx?id=31400&pname=file&aid=5663

Taentzer, G., Goedicke, M., & Meyer, T. (2000). Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proceedings, theory and application of graph transformations* (pp. 179–193). *Lecture Notes in Computer Science*, 1764. New York: Springer.

Voas, J. (1998, June). Certifying off-the-shelf software components. *IEEE Computer*, 31(6), 53–59. Retrieved April 9, 2007, from http://ieeexplore.ieee.org/iel4/2/15014/00683008.pdf?arnumber=683008.

Voas, J. (2000, August). Developing a usage-based software certification process. *IEEE Computer*, 33(8), 32–37. Retrieved April 9, 2007, from http://ieeexplore.ieee.org/iel5/2/18714/00863965.pdf?tp=&isnumber=&arnumber=863965.

Wermelinger, M., & Fiadeiro, J.L. (2000, March). A graph transformation approach to software architecture reconfiguration. In H. Ehrig & G. Taentzer (eds.), *Joint APPLIGRAPH/GETGRATS workshop on graph transformation systems* (GraTra'2000), Berlin, Germany. Retrieved April 9, 2007, from http://tfs.cs.tu-berlin.de/gratra2000/.

Wermelinger, M., & Fiadeiro, J.L. (2002). A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2),133–155.

Whaley, J., Martin, M.C., & Lam, M.S. (2002). Automatic extraction of object-oriented component interfaces. In *Proceedings of ISSTA'02* (pp. 218–228). New York: ACM Press. Retrieved April 9, 2007, from http://suif.stanford.edu/papers/whaley02.ps.

Xie, F., & Browne, J.C. (2003). Verified systems by composition from verified components. In *Proceedings of FSE'03* (pp. 277–286). New York: ACM Press.

Xie, G., & Dang, Z. (2004). An automata-theoretic approach for model-checking systems with unspecified components. In Proceedings of *FATES'04*. *Lecture Notes in Computer Science.* New York: Springer.

Zheng Li, Jun H., Yan J. (2005). Pattern-based specification and validation of web services interaction properties. In *Proceedings of the third international conference on service-oriented computing. ICSOC* 2005. New York: Springer, *Lecture Notes in Computer Science*, 3826, 73-86.

2003 - 2006 Sponsored Acquisition Research Topics

Acquisition Management

- Software Requirements for OA
- Managing Services Supply Chain
- Acquiring Combat Capability via Public-Private Partnerships (PPPs)
- Knowledge Value Added (KVA) + Real Options (RO) Applied to Shipyard Planning Processes
- Portfolio Optimization via KVA + RO
- MOSA Contracting Implications
- Strategy for Defense Acquisition Research
- Spiral Development
- BCA: Contractor vs. Organic Growth

Contract Management

- USAF IT Commodity Council
- Contractors in 21st Century Combat Zone
- Joint Contingency Contracting
- Navy Contract Writing Guide
- Commodity Sourcing Strategies
- Past Performance in Source Selection
- USMC Contingency Contracting
- Transforming DoD Contract Closeout
- Model for Optimizing Contingency Contracting Planning and Execution

Financial Management

- PPPs and Government Financing
- Energy Saving Contracts/DoD Mobile Assets
- Capital Budgeting for DoD
- Financing DoD Budget via PPPs
- ROI of Information Warfare Systems
- Acquisitions via leasing: MPS case
- Special Termination Liability in MDAPs

Logistics Management

- R-TOC Aegis Microwave Power Tubes
- Privatization-NOSL/NAWCI
- Army LOG MOD
- PBL (4)

- Contractors Supporting Military Operations
- RFID (4)
- Strategic Sourcing
- ASDS Product Support Analysis
- Analysis of LAV Depot Maintenance
- Diffusion/Variability on Vendor Performance Evaluation
- Optimizing CIWS Lifecycle Support (LCS)

Program Management

- Building Collaborative Capacity
- Knowledge, Responsibilities and Decision Rights in MDAPs
- KVA Applied to Aegis and SSDS
- Business Process Reengineering (BPR) for LCS Mission Module Acquisition
- Terminating Your Own Program
- Collaborative IT Tools Leveraging Competence

A complete listing and electronic copies of published research within the Acquisition Research Program are available on our website: www.acquisitionresearch.org

Acquisition research Program
Graduate school of business & public policy
Naval postgraduate school
555 DYER ROAD, INGERSOLL HALL
MONTEREY, CALIFORNIA 93943

www.acquisitionresearch.org