Efficient and Effective Testing of Multiple COTS-Intensive Systems

Dr. Richard Bechtold
Abridge Technology

Systems & Software Technology Conference
Thursday, 22 April 2004
Track 5: 1:30 - 2:15
Room: 150 G

Testing any complex system is a challenge, but testing multiple commercial off-the-shelf (COTS)-intensive systems is especially challenging due to a variety of factors. These factors include (1) systems combining highly diverse technology levels, (2) organizational users spanning numerous skill levels, (3) incorporating wide varieties of COTS vendors and packages, and (4) updated systems arriving for testing from projects that already may be substantially behind schedule and over budget. Regardless of these challenges, it is still essential to conduct efficient and effective testing in the context of limited time and nominal budgets. This article explains how to analyze, prioritize, plan, and manage the testing of COTS-intensive systems. Priorities are discussed from the perspective of identifying general types of transaction dialogues and specific instances of key dialogues. Dialogue value escalation is explained. Test planning and management are then discussed with the goal of maximizing the delivered benefit to organizational end-users.

Many organizations depend critically upon the maintenance and evolution of multiple in-house commercial off-the-shelf (COTS)-intensive systems. Most COTS systems are not matched perfectly to these organizations' specific needs and hence companies are often adjusting parameters, creating scripts, developing macros, writing code, or otherwise changing or extending functionality. Additionally, COTS vendors are always developing new packages, adding additional capability, and releasing updated versions of their products. Then, of course, there are the usual service packs, security patches, and other bug fixes that vendors want companies to install. And do not forget ongoing hardware and network upgrades. All this translates into a significant challenge for those responsible for performing testing to ensure adequate quality within this complex infrastructure.

For most organizations, testing everything is simply not an option. Hence, somehow testing must be biased in a manner that results in the maximum reduction of organizational risk, and that helps ensure the delivery of maximum benefit to system end-users. Amazingly, approximately half of all software modules in a system are defect-free and do not require any testing at all. Further, 80 percent of the defects come from 20 percent of the modules, and 90 percent of system downtime comes from at most 10 percent of the defects [1].

The potential exists to substantially improve upon system uptime and overall end-user benefit by finding, for example, just five percent more defects than are currently found. Ironically, end-user benefit can even be improved by finding *fewer* defects. This might occur if the testing approach is adjusted to focus on finding defects that cause the greatest system downtime. For example, the user experience will likely be substantially better if one defect is found that would crash the system daily versus finding three defects that would each crash the system annually.

Given the preceding data, a significant majority of testing efforts should be focused on 10 percent to 20 percent of any particular COTS system. The question is, how do we find that 20 percent? And since some in-house systems are clearly more critical than others, how do we account for that?

This article briefly identifies these and similar major challenges associated with performing efficient and effective testing of multiple in-house COTS-intensive systems. Then a nine-step process is explained that can help improve the approach to performing testing. The steps are as follows:

1. Identify Transaction Dialogues.
2. Analyze User Impact.
3. Determine Historical Defect Prevalence.
4. Prioritize Dialogues.
5. Prioritize Test Types.
6. Build Test-Dialogue Matrix.
7. Analyze Test Resources.
8. Establish a Test Plan.
9. Conduct Test Management.

Although the above sequence of actions may, at first, seem a bit daunting, several of the steps become substantially simpler after the first planning iteration. For example, once you have prioritized the types of tests to perform, you can typically reuse the same prioritization during future planning iterations. Additionally, this method is self-correcting. In earlier planning iterations, it is likely that you will have to rely upon a lot of guesswork to perform some of the steps. However, as you collect data during subsequent testing cycles, you can start using that data to replace, or at least augment, estimates. Note that the overall testing approach described in this article can also be applied to the testing of virtually any complex system, or for performing testing in environments where test resources are very scarce or test urgency is very high. However to simplify this discussion, the remainder of this article specifically focuses on discussion and examples relevant to testing multiple COTS-intensive systems.

COTS Testing Challenges

Testing of any complex software system can be challenging, and testing of COTS systems even more so because they require testing someone else's software (and hardware) and also testing your implementation and/or extension of that system. Deploy a few dozen COTS-intensive systems into the organization, and you have created a full-scale testing nightmare.

When performing testing in this type of environment, typically you need to consider and address most or all of the following issues:

- Different COTS systems may be based upon significantly different technologies.
- System end users usually have substantially different skill and experience levels.
- Your organization may depend upon a wide variety of COTS vendors.
- The threat levels presented by various COTS systems relative to core mission or business processes can differ by

orders of magnitude.

- System test work requests are not evenly distributed over time, and multiple parallel systems under test may rapidly overwhelm available test resources.
- Systems often become available for testing substantially later than originally planned.
- For some system failures, finding the actual source of the failure – or even re-creating the failure – can be quite complex and time-consuming, especially when the COTS system involves products from multiple vendors.
- The end-user organization may be reluctant to provide, or is incapable of providing (due to security reasons), actual production data for use in testing.
- Funds or resources originally allocated for system testing are often used to cover cost overruns during earlier project phases.
- System release or *go live* dates are often unmovable, or at least quite painful to move.
- Test organization resources are often redirected to perform unplanned emergency testing of systems to investigate problems reported by end users working with production systems.
- Calculating *return on investment* for software testing can be extremely difficult, or impossible [2, 3].

In summary, efficient and effective testing of multiple in-house systems requires a thorough understanding of the role those systems play and the value they provide – and the threat they present to the organization. It also requires a highly flexible plan to accommodate potentially significant organizational dynamics. The following nine-step process addresses these issues and is based upon these fundamental principles: (1) collect intelligence, (2) prioritize objectives, (3) understand resources, and (4) go for maximum impact.

1. Identify Transaction Dialogues

The COTS system value and impact analysis commences with identifying and documenting transaction dialogues. A transaction dialogue is a closely related set of actions – ideally mapped to one or more system requirements – that are normally performed at the same time, and that typically conclude at the same place they commenced. For example, adding new employees to the employee database can be viewed as a dialogue. It likely starts and ends on the same menu screen, and the data entry actions for adding employee information are closely related.

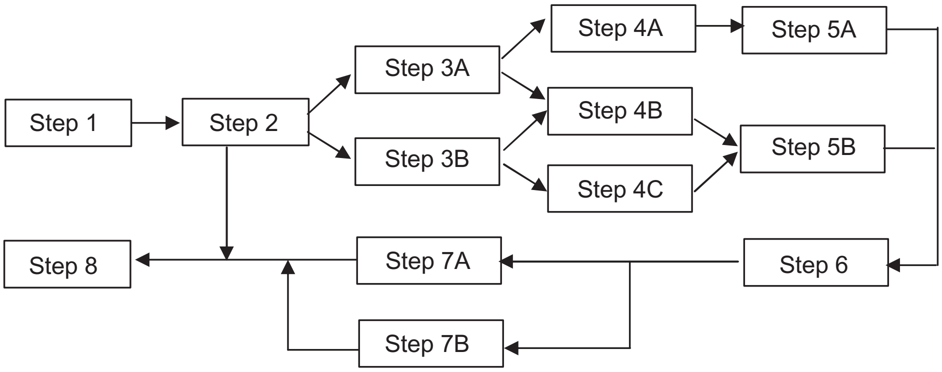

Figure 1: *Identifying and Diagramming Transaction Dialogues*

Another example of a dialogue is updating electronic timesheets with the activities performed and hours worked during a given day. Further examples include entering sales orders into an accounts receivable system, entering paycheck information into a payroll system, or paying invoices within an accounts payable system. When creating transaction dialogues, remember that the ultimate objective is to structure or design them so that they directly facilitate testing. (Note: If you already have a set of use-case scenarios [4], you can use those in lieu of creating transaction dialogues. In the absence of use-case scenarios, defining dialogues will likely be faster and easier than defining scenarios).

Dialogues can be easily diagrammed as a directed graph consisting of a set of nodes and arcs where the nodes represent specific (and possibly complex) activities or steps, each with a descriptive label and possibly a brief description. The arcs, each of which connects two nodes, indicate the sequence in which these activities can occur. As shown in Figure 1, multiple arcs departing from a node indicate that any of the destination actions might occur next. Similarly, multiple arcs arriving at a node indicate any of the source activities may have occurred previously. Note that there are no semantic differences between two arcs departing (as in Step 2) or in one arc departing and splitting (as in Step 6). Likewise there is no difference between multiple arcs arriving (as in Steps 4B and 5B) or multiple arcs joining and then arriving as one (as in Steps 6 and 8).

In order to ensure clarity and reduce misunderstanding, it is important to keep these drawings, and their associated semantics or rules, simple (i.e., these diagrams do not require the formality or detail of state transition diagrams). The objective of developing these diagrams is just to clearly indicate the boundary of the closely related set of system and user actions or steps that are covered by the dialogue.

It may be infeasible or unnecessary to try to capture all the various dialogues within a system. Hence, develop dialogues in a somewhat prioritized order. That is, what are the most important things a given system does? Capture that at a high level. Then, resources permitting, decompose complex nodes on higher-level dialogues into their own dialogues. If you are preparing to perform parallel testing of multiple COTS systems, then continue with this strategy by documenting the most important dialogues in the most important COTS systems, etc.

Again, resources and time permitting, steadily extend both coverage and formality by adding dialogues related to less-important systems and less-important system interactions, and by adding additional information to previously developed higher-priority dialogues. For the most important dialogues, consider augmenting them with comprehensive use-case scenarios.

When identifying dialogues within any given system, the ultimate objective is to gain a better understanding of where, how, and why users interact with that system. This sets the foundation for analyzing user impact.

2. Analyze User Impact

As you build a collection of transaction dialogues, you can begin to identify specific *user groups* within the end-user organization. Note that the groups you identify may not align with actual job titles held by the end-users. This is common and acceptable because your primary objective in this step is to improve your understanding of how closely related sets of people – regardless of their titles – interact with the various systems during day-to-day usage.

Depending upon the complexity of the end-user organization(s), you may also find it convenient to define a set of user-group types. Type distinction might be quite simple such as distinguishing between *beginning*, *intermediate*, and *advanced* users. For each identified user-group (or group-type combination), assign a code to indicate that group's estimated contribution to core mission or business performance. For example, use A to indicate a group that is critical to core performance, B to indicate a group that is important, and C to indicate a nonessential support group.

Next, develop estimates of the number of users within each group. Typically the sum of this number across all user-groups will be much larger than the total number of users because some people will belong to multiple groups within one system. Additionally, numerous people or organizations may interact with multiple systems.

The last and possibly most difficult part of this step is to generally estimate the number of times an average user invokes a given dialogue. For example, an average user within a particular group might invoke one dialogue an average of once per month, and another dialogue 20 times per day. Ideally, you can collect feedback from system end-users to help you develop these estimates. Alternatively, you may need to resort to best-guess estimates developed by you, your test team, and any available systems analysts.

The final objective of this step is to have some idea regarding the relative frequency that various dialogues are occurring. This data will eventually be used to help prioritize dialogues, but that prioritization will also depend upon a dialogue's historical defect profile, so that step must be taken next.

3. Determine Historical Defect Prevalence

For each dialogue, document the average number of defects normally found during testing, and add the average number found after testing (that is, failures reported by system users). This is a relatively simple step; you will either have this data or you will not. Probably, you will not. Nevertheless, if you have been in the test group for a while you can likely invent some reasonably usable numbers. Alternatively, you may be able to use historical data from similar dialogues elsewhere within the system. These numbers do not need to be either accurate or precise. All we are trying to determine is which dialogues tend to contain the most defects. Whether a given dialogue has been 10 times more defective than another or 20 times more defective is not really important. For the prioritization step, it is sufficient to know that it is normally *much* more defect prone.

4. Prioritize Dialogues

Eventually, we are going to be testing based upon dialogues (and possibly use-case scenarios). The question at this point is which ones and how thoroughly? As indicated previously, the overall objective is to perform efficient and effective testing of multiple in-house COTS-intensive systems by striving to perform testing in a manner that reduces organizational risk and that helps ensure the delivery of maximum benefit to the end-users. Given this objective, relative importance can be established by examining the following for each dialogue:

- The size of the user-groups that invoke the dialogue.
- The frequency of invocation per unit time (e.g., 50 users, 20 invocations per user per day).
- The relative role of the user-group and dialogue to the organization's core mission capabilities.
- The historical defect prevalence associated with the dialogue.

Note that this step is an analytical step, not a calculation. The data developed in prior steps is used to help make highly informed decisions regarding how to prioritize dialogues. However, remember that much of the data may be gross estimates at best and some may be outright speculation, especially during early planning iterations. Hence, use the underlying numbers as an aid and combine them with the experience and judgment of you and your team.

The ideal output for this step is a lifeboat ranking (or a *fully ordered set*) of all the dialogues that are candidates for the next phase of testing. Basically, you start at the top of the list and keep working downward as far as you can go, resources permitting. If a fully ordered approach is infeasible in your environment, then strive to arrange dialogues into prioritized testing sets (e.g., *critical, highly important, important,* etc.).

5. Prioritize Test Types

This step consists of determining the types of tests to run relative to the dialogues. That is, are you only interested in testing features, or do you also consider other tests to be important? Examples of other types of tests include the following:

- Performance (normal load).
- Stress (approaching design limits).
- Overload (substantially beyond design limits).
- Security.
- Safety.

The purpose of this step is to ensure that everything that is important regarding organization risk management and end-user benefit is considered. For some environments the prioritization of which types of tests should be run first is fairly stable. For example, in life-rated systems, safety is typically the highest priority. After this step, you have everything necessary to build the test dialogue matrix.

6. Build a Test-Dialogue Matrix

This is another potentially simple step where you build a matrix to help ensure that all appropriate tests for each of the designated dialogues are run. For example, although you may be planning to conduct testing focused specifically on security, there may be numerous dialogues where security is not an issue and security testing is not required. Within this matrix (dialogues, in priority order, shown as rows; and test types, also in priority order, shown as columns), simply mark any cell to indicate that you want to perform that particular test (column) on that particular dialogue (row).

7. Analyze Test Resources

The last step to perform before building your test plan is to consider the resources available for testing. This includes not only human resources but also any automated tools you have that can be used to help with any subset of the testing. The general approach is to run the most important tests and to cover the maximum amount of highest priority dialogues. However, the overall objective continues to be risk reduction and user benefit, so it might make sense to adjust the order of the tests because you have one or more automated tools that allow you to rapidly and effectively test numerous lower priority dialogues. This would be the preferred approach when the compound benefit of testing a large number of lower priority dialogues exceeds that of the delayed higher-priority tests.

Integral to this approach is the fact that it is often difficult, if not impossible, to know your test resource availability relative to future demands that may be placed on your test group. This is why each of these steps is ultimately about prioritization and best utilization of available resources. For example, you may think you still have a month to finish testing the dozen COTS systems that are currently under test, but then again, what if you do not? Especially when it comes to the highly dynamic environments typically associated with testing multiple COTS systems, it is imperative that whatever is most important to occur next (whether it is a single human-intensive test or a highly automated set of tests) is precisely what occurs.

8. Establish a Test Plan

Dynamic environment notwithstanding, it is important to take the time to develop an actual test plan, or to post your latest updates and revisions to the existing test plan. The formality and scope of various test plans may vary significantly depending on system criticality (or lack thereof), your overall test strategy, and resource availability. For example, there undoubtedly will be situations where exploratory testing [5] will give far better coverage (and test resource utilization) than scripted testing; additionally, you minimize the creation of comprehensive test documents to maximize the performance of actual test activities. Similarly, you will likely want to consider some level of combinatorial testing [6] to reduce the overall number of required tests by focusing on two-way and three-way parameter combinations for effectively finding multi-mode software faults. Regardless of the overall strategy, however, in the interests of communication, consensus, and overall risk management, some level of documented test plan is almost always beneficial.

In many environments, test plans (formal or otherwise) are frequently overtaken by events and therefore need frequent revision. For example, you may want to review your test plan on a weekly or monthly basis to determine if any revisions are required. Your test schedule, however, including revisions to dialogue prioritization, might need to be updated on a daily or weekly basis.

When planning, consider the various development groups within your organization, and absolutely consider their track record regarding their commitments to the test group. Do they normally deliver system and software enhancements on time, or do they routinely miss by a few months? The problem is, although you have to consider this when preparing your test schedule, project managers might find it a tad offensive if your schedule shows, for example, you are not really expecting them to deliver until six months after their claimed completion date. Therefore, consider keeping both an "official test schedule" and a "pessimistic test schedule." The pessimistic schedule is what you use to remind yourself how you think events are really going to unfold.

As you are building or revising your plan and schedule, always consider the possibility of obtaining additional test resources. For example, can developers help with system testing? Absolutely. Can system designers test? Certainly. Are systems engineers, requirements analysts, and end users all potentially effective testers? Of course they are. Are any of these resources available to you? They probably are not. But then again, maybe that depends on the relative importance of the items on the upcoming testing schedule relative to some of the other work currently occurring within your organization. Given the steps you have taken to create the inputs to your multi-system test plan and schedule, if you think additional resources are needed, at a minimum you can support your request with a very compelling rationale.

9. Conduct Test Management

> *"The potential exists to substantially improve upon system uptime and overall end-user benefit by finding, for example, just five percent more defects than are currently found."*

Effective test management requires a wide variety of skills and activities, including the identification, collection, and analysis of a variety of test-related and quality-related metrics, and metrics associated with test status tracking, management, and control; proper reviews (to varying levels of formality) of test documentation and support material; and the determination of clear criteria for objectively assessing whether or not a system is ready for piloting, and when it is ready for operational use. Within this context, test activities should be prioritized with the ultimate objective of delivering maximum benefit to the end-users.

Prioritization is also influenced by the overall test strategy (such as exploratory testing and/or combinatorial testing). As testing activities are performed, there is normally a wealth of defect data collected. However, much of this strategically valuable data often is eventually lost. That is, test results are compiled and sent back to the development teams and are used for system and software corrections, but that data is not otherwise archived and managed by the test organization.

Note that one of the critical steps previously described is the determination or estimation of defect prevalence by dialogue. Since you are running the tests, you absolutely can and should retain this information. Ideally, you have (or can find or create) one or more failure-related reference lists so that system failures can be assigned to failure categories, failure severities, failure likelihood in the field, etc. This data can then be stored in a database, or even in a spreadsheet for use during future planning iterations.

Possibly even more important, to continue to enhance this overall process you must collect relevant data regarding failures or problems reported from system beta testing and production usage. These are the defects that slipped past your testing process. What are they? Where are they? When were they discovered? How were they discovered? For a given defect, which user group reported it first? Which user group reported it the most? What was the impact? This data is absolutely essential to you in your ongoing efforts to understand end-user consequences and organizational risk, and to continually improve the contributions of your test organization to overall mission success.

Delivering Maximum Benefit

Delivering maximum benefit to organizational end-users, and ultimately to the organization, is everyone's job. Therefore, related to this discussion are numerous quality topics such as the value of peer reviews, the benefits of early inspections, the greater benefits of perspective-based reviews [1], etc., which directly impact overall organizational competence and mission performance. However, as a manager or technical specialist within the test organization, you normally do not influence such things. The projects build what they build, the procurement organization buys what it buys, and the maintenance teams implement changes however they feel like. And you are the last line of defense, literally, for your organizational system's end-users.

There are many alternative approaches to testing COTS-intensive systems. You can, for example, assign functional requirements to different priority levels, and then take a priority-driven approach to test performance. Alternatively, many organizations, if they have sufficient resources, plan and perform tests by taking one of these approaches: *try-to-test-everything-once*, *top-down*, *front-to-back,* or any of numerous other techniques.

The premise behind the approach described in this article is to increase your efforts relative to test priority analysis and planning, and to strive to maximize the test organization's contribution to the overall delivery of benefits to systems' end-users, and the overall end-user experience. Invariably, testing priorities will continue to be a constantly shifting landscape due to unexpected events. However, by regularly following the steps described in this paper you will steadily improve your understanding of user-group interactions with various systems and steadily increase the amount of data available to you related to user-discovered defects and consequences. This combination of strategy, understanding, and data will directly support and enhance your overall efforts toward leveraging your test organization to deliver maximum benefit to your end-users.◆

About the Author

Richard Bechtold, Ph.D., is president of Abridge Technology and is an independent consultant who assists industry and government with organizational change and systematic process improvement, especially in the area of implementing effective project management. Bechtold has more than 25 years of experience in the design, development, management, and improvement of complex software systems, architectures, processes, and environments. This experience includes all aspects of organizational change management, process appraisals, process definition and modeling, workflow design and implementation, and managerial and technical training. Bechtold is an instructor at George Mason University where he teaches graduate-level courses in software project management, systems analysis and design, principles of computer architectures, and object-oriented java programming. The second edition of his latest book, "Essentials of Software Project Management," is scheduled for publication in 2004.

Abridge Technology
42786 Oatyer CT
Ashburn, VA 20148-5000
E-mail: rbechtold@rbechtold.com

References

1. Boehm, B., and V. Basili. "Software Defect Reduction Top-10 List." NSF Center for Empirically Based Software Engineering, Jan. 2001 <www.cebase.org/www/home/index.htm>.
2. Bechtold, R. "Software Quality Valuation: Return on Investment Versus Reduction of Risk." International Conference on Practical Software Quality Techniques/Practical Software Testing Techniques. Washington, D.C., June 2003.
3. Bechtold, R. "Return on Investment for Software Testing." Software Testing Forum. Reston, VA., Mar. 2003.
4. Satzinger, J., and T. Orvik. The Object Oriented Approach, Concepts, System Development, and Modeling with UML. 2nd ed. Course Technology, Thomson Learning, 2001.
5. Bach, J. "What Is Exploratory Testing?" <www.satisfice.com/articles/what_is_et.htm>.
6. Daich, Gregory T. "No-Cost Combinatorial Testing Support." Software Technology Support Center, Hill AFB, UT., <www.stsc.hill.af.mil/consulting/sw_testing/improvement/cst.html>.

Web Sites

Object Management Group

www.omg.org
Founded in 1989 by 11 companies, the Object Management Group (OMG) now has about 800 members. It is a not-for-profit corporation formed to create a component-based software marketplace by accelerating the introduction of standardized object software. The OMG is establishing the Model Driven Architecture through its worldwide standard specifications, including Object Services, Internet Facilities, Domain Interface specifications, and more.

The Agile Alliance

www.agilealliance.com/home
The Agile Alliance is a non-profit organization dedicated to promoting the concepts of agile software development, and helping organizations adopt those concepts. The site features an extensive library of articles about agile processes and agile development.

Center for Software Engineering

http://sunset.usc.edu/index.html
Dr. Barry Boehm founded the Center for Software Engineering (CSE) in 1993. It provides an environment for research and teaching large-scale software design and development processes, generic and domain-specific software architectures, software engineering tools and environments, cooperative system design, and the economics of software engineering. One of CSE's main goals is to research and develop software technologies that can help reduce cost, customize designs, and improve design quality by doing concurrent software and systems engineering.

INCOSE

www.incose.org
The International Council on Systems Engineering (INCOSE) was formed to develop, nurture, and enhance the interdisciplinary approach and means to enable the realization of successful systems. INCOSE works with industry, academia, and government to disseminate systems engineering knowledge, promote collaboration in systems engineering, establish integrity in systems engineering standards, and encourage research and educational support to improve the systems engineering process and its practices.

Risk Management

www.acq.osd.mil/io/se/risk_management/index.htm
This is the Department of Defense (DoD) risk management Web site. The Systems Engineering group within the Interoperability organization formed a working group of representatives from the services and other DoD agencies involved in systems acquisition to assist in the evaluation of the DoD's approach to risk management including the latest tools and advice on managing risk.