A Framework for Evaluating Software Technology

ALAN W. BROWN and KURT C. WALLNAU, Software Engineering Institute

Many organizations struggle to make informed decisions when investing in new software technologies. The authors' experimental framework can help companies evaluate a new software technology by examining its features in relation to its peers and competitors through a systematic approach that includes modeling and experiments.

Software development organizations continually make decisions regarding software technology selection, application, and introduction. Companies decide on some technologies explicitly after examining alternatives in detail, such as deciding on a standard word processor for the organization. Companies decide on other technologies implicitly with little study of the decision's potential impact: for example, deciding to ignore a new product line from an operating system vendor. In both cases, the organization attempts to understand and balance competing concerns regarding the new technology. These concerns include the
- initial technology acquisition cost;
- long-term effect on quality, time to market, and cost of the organization's products and services when using the technology;
- training and support services' impact of introducing the technology;
- relationship of this technology to the organization's future technology plans; and
- response of direct competitor organizations to this new technology.
On the basis of such factors, the organization assesses the technology's likely return on investment. In domains outside of software technology, ROI is well defined and calculated by established formulas. Attempts to calculate software technology ROI, however, have generally failed. The foremost reason for this is the difficulty in establishing cause and effect when assessing new software technologies' impact on an organization.

Instead of a concrete ROI prediction for a new technology, most organizations obtain an informed view of the technology from applying informal techniques (for example, attending trade shows, designing pilot applications, conducting case studies). Although informal, the techniques nevertheless lead to an "informed intuition" about the technology on which to make a decision.

Performing technology evaluations is integral to the Software Engineering Institute's role. The SEI advises customers on current software engineering best practices, helps to adopt promising technologies, and raises awareness of technology trends. One particular SEI effort is STEIM (software technology, evaluation, integration and measurement), which focuses on developing evaluation techniques and measures that apply to technologies supporting system integration of off-the-shelf components. Our STEIM experiences highlight the need for greater rigor in evaluation data collection and analysis, and for more systematic evaluation planning, execution, and interpretation.

To address these evaluation problems, we have drawn upon our evaluation experiences to develop a conceptual framework that helps to categorize different software technologies and suggests a way for organizations to systematically evaluate new software technology. The premise of our framework is that an organization requires qualitative and quantitative reports on technology "deltas"—descriptions of how a new technology's features differ from those found in existing technologies.

This framework is a distillation of our experiences with software technology evaluation over the past five years. It is our attempt to make the process more systematic and repeatable. In this article we describe our framework, explain how it highlights techniques to identify technology deltas, and define experiments to estimate the costs and benefits of these deltas in specific usage scenarios. We then apply the framework to a specific technology example in the problem domain of component integration. Our technology delta framework

- sets technology evaluation goals based on understanding the added value of a new technology;
- uses different evaluation techniques in an overall framework to synthesize disparate results obtained; and
- facilitates individual product evaluations that concentrate on their distinguishing characteristics in relation to their technology precursors and product peers.

**CURRENT EVALUATION APPROACHES**

Most organizations recognize the importance of technology enhancement to improve the quality of their products and services, to be competitive, and to remain attractive both to investors and to a technology-oriented workforce. Careful decision making on new (or updated) technologies is thus essential, as is the timely, balanced information that informs those decisions.

**Technology evaluation in practice.** Technology evaluations are generally ad hoc, heavily reliant on the evaluation staff's skills and intuition. We've identified the following approaches currently practiced, in various combinations, by software development organizations in evaluating new technologies:

- obtain objective technology data by documenting case studies at other organizations;
- gather subjective opinions and experiences with the technology by attending trade shows and by conducting interviews with, or sending questionnaires to, technology vendors and users;
- conduct focused experiments to mitigate high-risk aspects;
- demonstrate the technology's feasibility with a pilot project;
- compare the technology to existing practices by conducting a shadow project and examining the results of both approaches; and
- initiate demonstrator projects in the organization to acquire phased exposure to the technology.

What is missing from all these approaches is a well-developed conceptual framework that lets the evaluation results be considered in terms of what this new technology adds to the organization's existing technology base. Instead, a typical organization applies one or more of the approaches and, on the basis of the resultant data, forms an intuition about that technology's value. Much of the informality in interpreting any evaluation's results is due to the absence of:

- well-defined goals before starting the evaluation;
- controlled, rigorous techniques for data gathering during the evaluation;
- a conceptual framework for ana-
lyzing the resultant data in the context of existing technologies.

To date, practitioners have concentrated chiefly on data-gathering techniques.\textsuperscript{1}\textsuperscript{4}

**Technology evaluation in the literature.** In analyzing software engineering's technology evaluation literature, we can distinguish between two classes of evaluation:

- **Product-oriented:** selecting among a set of products that provide similar functionality (for example, a new operating system, design tool, or workstation);
- **Process-oriented:** assessing the impact of a new technology on existing practices to understand how it will improve performance or increase quality (for example, a new design methodology, programming language, or software configuration management technique).

Many organizations have relatively mature evaluation techniques for the product-oriented decisions, and this is reflected in the literature. For example, the International Standards Organization cites numerous descriptions of general product evaluation criteria,\textsuperscript{1} while others describe specialized techniques that consider the needs of specific application domains (for example, CASE tools).\textsuperscript{6}

Organizations facing process-oriented evaluation decisions have a more difficult task. Here, an organization tries to assess the potential impact of a new technology and estimate its impact on the organization's practices, products, and services. The literature tends to focus on process improvement first, which is often based on the SEI's Capability Maturity Model (CMM) or on ISO 9000 standards, with technology support a secondary consideration.

Interesting exceptions to this approach include Germinal Boloix's and Pierre Robillard's system evaluation framework that quickly, in about an hour and a half, provides high-level information to managers about a software system's characteristics.\textsuperscript{7} The strength of their framework is that it offers a broad system snapshot by considering the perspectives of end users, developers, and operators. Little detailed insight into the strengths and weaknesses of a technology, however, in comparison with its peers is either sought or revealed.

A second exception is the work by Tilman Bruckhaus, who defined a technology impact model and applied it to a large CASE tool development, as described elsewhere in this issue. The method supplies quantitative data on the impact of various CASE tool alternatives for a particular scenario. While the approach is valuable, it concentrates only on impact as either increasing or decreasing the number of process steps, ignoring the intrinsic value of the technology itself and avoiding any attempt to compare peer technologies for their relative added value.

**FRAMEWORK PRINCIPLES AND TECHNIQUES**

The key idea behind our technology delta framework is that technology evaluation depends on two factors:

- understanding how the evaluated technology differs from other technologies;
- understanding how these differences address the needs of specific usage contexts.

Our emphasis is on developing rigorous techniques to address both. These techniques include informal descriptive modeling techniques for documenting assertions about the nature of a technology and its usage context, and more empirical techniques for conducting experiments.

Determining a technology's added value means identifying the features that differentiate it from other technologies, and evaluating these differential features—the feature delta—in a well-defined application context. How can you identify and then assess feature deltas in a disciplined way? By following the three phases of our technology delta framework—descriptive modeling, experiment design, and experiment evaluation.

**Technology delta principles.** The framework and its phases embody four important tenets. First is the belief that a technology's potential impact is best understood in terms of its feature delta. All of a technology's features are relevant to understand the technology and its application, but for you to understand its added value, you must focus on its distinctive features in relation to other technologies.

Second, a technology's distinctive features must be described, then evaluated in a sharply focused usage context. With the framework, the technology evaluator forms hypotheses about how a feature delta supports a defined usage context, and employs rigorous experimental techniques to confirm or refute these hypotheses.

Third, the framework reflects and embraces the inherent complexity, ambiguity, and dynamism of the technology marketplace. A technology's features reflect both the way a technology is meant to be used and the features offered by competing technologies: Both change over time. Descriptive

**A technology's distinctive features must be described, then evaluated in a sharply focused usage context.**

modeling, for analyzing and documenting the interdependencies between technologies, and between technologies and their usage contexts, is crucial to a disciplined technology evaluation.

Last, the framework reflects a limited but well-defined objective, specifically, determining added value. Factors
to consider when you evaluate a technology of course include installation costs, market forecasting, organizational resistance, and other nontechnical considerations. However, these nontechnical considerations are not the focus of our framework.

Figure 1 is a high-level depiction of the technology delta framework. Note the three phases—descriptive modeling, experiment design, experiment evaluation—and their outcomes.

Descriptive modeling phase. Descriptive models are descriptions of assumptions made by the technology evaluator concerning features of interest and their relationship to usage contexts. These assumptions will later form the basis of experimental evaluation. The descriptive models are a foundation for a rigorous approach to describing technologies, for achieving consensus on the key features needed to distinguish technologies, and for documenting the evaluation process itself.

The descriptive modeling phase addresses feature discovery and impact prediction through the development of technology genealogies (ancestry of the technology) and problem habitats (uses of a technology, and its competitors), respectively. The technology genealogy reflects that new technologies are most often minor improvements of existing technologies. To understand a technology's features, we must also understand that technology's historical and technological antecedents. For example, to understand the unique contribution of object-oriented programming technology, we must understand its heritage in programming language theory (data abstraction, polymorphism), design theory (information hiding), and simulation and human-computer interaction (modeling real-world entities in software).

Technology features alone, however, are insufficient for us to understand added value; for this, we must understand how these features will be used and what benefits will accrue from their use—this is the role of the problem habitat. For example, the World Wide Web essentially integrates several preexisting technologies (Internet, graphical user interface, scripting language, protocols, and hypertext), and can be described largely in terms of the features of these constituents. The potential impact of the Web, however, belies its modest lineage. To predict this impact, we must understand the Web in terms of its potential use (its habitat) in areas such as electronic publishing, entertainment, and direct sales.

The output of this descriptive modeling phase is a situated technology: models that describe how a technology is related to other technologies, and the usage contexts in which it can be evaluated. Situating a technology in a technology marketplace and in a problem domain gives us a basis for identifying feature deltas.

In building our framework, we borrowed descriptive modeling concepts, such as domain genealogy and comparative feature analysis, from organizational domain analysis. Unlike domain analysis, however, the descriptive models generated for technology deltas are not end products but guides for structuring evaluation experiments and for interpreting the results of these experiments.

Genealogies and habitats can be modeled as semantic networks—nodes and links, where nodes represent "concepts" and links represent relationships among concepts. Several different node and link types are useful and are summarized in Table 1; other node and link types may be defined by the analyst. Nodes in these models represent different views of technology, while the links help establish relationships between these views and form a basis for making assertions about feature deltas. It's also useful to annotate the links with feature lists. For example, consider a hypothetical genealogy of middleware technologies. In such a genealogy we could describe technologies such as Hewlett-Packard's SoftBench product and Steve Reiss's FIELD prototype as peers; this link could be annotated with the features that distinguish SoftBench from FIELD.

Experiment design phase. This phase is essentially a planning activity. As illustrated in Figure 1, three activities are involved: comparative feature analysis (referred to as "comparative anatomy" in Figure 1), hypothesis formulation, and experiment design.

Comparative feature analysis involves a more detailed investigation of feature deltas. We have found that as the technology evaluator forms hypotheses and designs experiments, questions arise that require a closer look at technology features; this closer look might suggest other hypotheses and approaches. Why isn't comparative feature analysis part of the descriptive modeling phase as another kind of fea-

Figure 1. Technology delta evaluation framework.
ture study? We included it here because we have empirical rather than purely descriptive techniques for conducting comparative feature analysis:

- reference model benchmarking for qualitative feature descriptions using an a priori feature vocabulary; and
- feature benchmarking for quantitative feature descriptions.

For our purposes, a reference model is an annotated feature list. In some software technology domains reference models have already been developed, with more internal structure than mere feature lists. A reference model provides a ready vocabulary of features and, sometimes, their interrelationships. By mapping peer technologies to these reference models using the profiling techniques we describe later, we can use a common vocabulary to describe features. Surprises sometimes result: For example, two competing technologies may be found to provide complementary rather than overlapping services; this might suggest compatibility experiments that address their combined use.

Feature benchmarks quantitatively measure features in terms that make sense for the feature and the technology but in a way that is not necessarily focused on a specific problem domain. To illustrate, for middleware products we might measure message throughput under various load conditions. Such benchmarks may represent weak hypotheses, that is, the kinds of load factors that will influence performance. As with reference model benchmarking, feature benchmarking may reveal properties of a technology that suggest hypotheses about its use in a particular problem setting. (We give examples of reference model benchmarking and feature benchmarking in a later section.)

In formulating a hypothesis, the technology evaluator must ensure that

- the hypothesis is refutable from experimental evidence;
- suitable experimental techniques exist to test it; and
- the set of hypotheses are sufficient for evaluating added value.

The first item requires discipline and precision on the part of the evaluator. The second item requires familiarity with various evaluation techniques. The third item is inherently difficult to validate; thus, to ensure completeness, the evaluator should establish traceability links from hypotheses to the problem domain, which suggests a fairly detailed model of the habitat.

The output of the experiment design phase is

- a set of hypotheses about the added value of a technology that can be substantiated or refuted through experimentally acquired evidence, and
- a set of defined experiments that can generate this evidence and that are sufficiently comprehensive to support sound conclusions regarding added value.

**Experiment evaluation phase.** This phase is where evaluators conduct experiments, gather and analyze experimental evidence, and confirm or refute hypotheses. We've begun to catalog different experimental techniques that, given certain kinds of hypotheses, requirements for precision, and budget considerations, can help an evaluator.

Model problems. These are narrowly defined problems that the technology can address. They can be extracted from broader considerations of an application domain. Examples of model problems might include determining schedulability in real-time and manufacturing domains and integrating applications with independent control loops in the component integration domain. Model problems offer a narrow evaluation context and let alternative technologies be directly compared in a way that might be too expensive to do in a broader setting (for example, through demonstrators). They can be derived from a number of sources identified in the genealogy and habitat. Problem contexts are also a ready source of model problems: interprocess communication and distributed systems problem contexts have many known model problems.

**Compatibility studies.** Rather than determining how a single technology behaves in a narrow problem context, compatibility experiments show whether technologies interfere with each other or whether they can be effectively combined. These experiments are particularly useful if a technology might be inserted into an established technology baseline where interactions between the new and established technologies are suspected. Compatibility experiments can also help determine if the disjoint features of competing technologies can be used without interference from overlapping features.

**Demonstrator studies.** There is no substitute for trial applications of a technology in a real-life, scaled-up application setting. Although full-scale demonstrator applications can be expensive, a properly designed demonstrator can achieve some of the scale factors needed to stress-test a technology under

---

**TABLE 1. PRIMITIVES FOR GENEALOGY AND HABITAT MODELS**

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Form</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>Node</td>
<td>A class of functionality</td>
</tr>
<tr>
<td>Specification</td>
<td>Node</td>
<td>A description of a technology for its producers/consumers.</td>
</tr>
<tr>
<td>Product</td>
<td>Node</td>
<td>An implementation of a technology or specification.</td>
</tr>
<tr>
<td>Peer</td>
<td>Link</td>
<td>Nodes (of any type) that have similar features.</td>
</tr>
<tr>
<td>Competitor</td>
<td>Link</td>
<td>Products or specifications that compete in the marketplace.</td>
</tr>
<tr>
<td>Problem context</td>
<td>Link</td>
<td>Class of problems addressed by a node (of any type).</td>
</tr>
<tr>
<td>Is-a</td>
<td>Link</td>
<td>Product or specification is an instance of a technology.</td>
</tr>
<tr>
<td>Part-of</td>
<td>Link</td>
<td>Bundled and/or separable products or specifications.</td>
</tr>
</tbody>
</table>

---

**FIGURE 1. MODEL PROBLEM GENEALOGY FOR A TECHNOLOGY**

- An implementation of a technology or specification.
- Nodes (of any type) that have similar features.
- Products or specifications that compete in the marketplace.
- Class of problems addressed by a node (of any type).
- An instance of a technology.
- Bundled and/or separable products or specifications.
To show our technology delta framework in action, we applied it to the Object Management Group's Object Management Architecture (OMA)\(^6\) and its more widely known component, the Common Object Request Broker Architecture (CORBA). With commercial implementations of OMA now available, experts in object-oriented technology, middleware technology, and operating systems are questioning the OMA's uniqueness and its effectiveness in problem domains. By investigating the OMA technology delta with our framework, we've come up with some answers to these questions. In this article, we concentrate on how the framework has been used and omit many of the detailed evaluation results, which are documented elsewhere.\(^7\)

Figure 2 shows the key OMA elements. CORBA is a communication infrastructure that lets clients locate and make requests of (possibly remote) objects. There are different classes of objects within the OMA. Common Object Services are objects that provide widely applicable services, such as transactions, event management, and persistence. Common Facilities are objects that provide useful but less widely used services, such as electronic mail. Finally, application objects are application-specific, and are not presently a subject for standardization within the Object Management Group. In the following discussion, "OMA" refers to all of the elements depicted in Figure 2, while "CORBA" refers only to the communications design and implementation, and that experiments from this phase yield not just hypothesis confirmation or refutation but insights into the optimal use of a technology to address these underlying critical issues.

**OMA/CORBA DESCRIPTIVE MODELS**

**Synthetic benchmarks.** Synthetic benchmarks help in examining technologies with runtime aspects and where the problem domain can be simulated. For example, in evaluating a middleware technology, test messages can be injected into a command and control system to simulate system operation in normal and crisis modes. Considerable effort may be required to acquire valid synthetic loads—for example, through instrumentation of existing applications. Synthetic benchmarks differ from feature benchmarks precisely in the degree to which such problem-domain-specific factors are included in the experiment.

**Benefits.** Besides data collection, the hands-on experimental approach helps the evaluator become competent with the technology being evaluated. Moreover, we've found that hypotheses often focus on critical usage issues, for example, investigation while excluding other factors. For example, system documentation could be dispensed with, or reliability and performance might be downplayed, assuming these were not crucial to any hypothesis. Nevertheless, demonstrators may require an organization to commit substantial resources, and this phase of an evaluation is generally deferred until a technology has shown reasonable prospects for success. A demonstrator study differs from a pilot project. Pilot projects are intended to be initial full deployments of a technology. In contrast, demonstrators may elide some otherwise essential engineering efforts. Moreover, demonstrators focus on a technology's distinguishing features such that a demonstrator may deliberately exaggerate some features in a problem setting to expose its strengths and deficiencies. In contrast, pilot efforts will be driven by optimal engineering trade-offs.

**Figure 2. The Object Management Architecture.**

**Figure 3. OMA/CORBA genealogy.**
OMA/CORBA genealogy. Figure 3 depicts part of the OMA genealogy, focusing on component integration concepts. To some extent, OMA and CORBA are separable technologies. The genealogy also indicates that the concepts (nodes) relating to OMA via "is-a" relationships are not necessarily technologies but suggest properties that may be significant in evaluating the OMA. To illustrate, the "OMA is-a technology specification" relationship implies, minimally, that OMA implementations must conform to an authoritative specification. The "PCTE is-a technology specification" also implies conformance rules. However, while both OMA and Portable Common Tools Environment (PCTE) specifications define notions of conformance, they do so in different ways.

Peer relationships are also indicated in Figure 3. Microsoft's Object Linking and Embedding is a peer technology to OMA, which implies that both technologies share some common features. The relationship does not imply, however, a substitutability relationship between OLE and OMA. This serves as a warning to analysts: The open-ended nature of genealogy and habitat models does not imply a lack of precision in the models—if the models are to be a rational foundation for experiments, the relationships must be documented and consistently used.

OMA/CORBA habitat. Figure 4 illustrates part of the OMA habitat that we described. Because a habitat is generally more complex than a genealogy, evaluators must be careful to balance thoroughness against complexity. Because the habitat is an assertion by the evaluator about the technology features that are of interest, the concepts that are missing from the habitat are just as significant as those that appear. Our work with evaluating OMA focused on its use to support component integration, interworking object technologies, and object-oriented systems development. Other possible problem contexts were thus disregarded.

The OMA habitat also illustrates that it's often useful to extend the model beyond the immediate technologies of interest to better understand the technology being evaluated. For example, the OMA and OLE are often regarded as competing technologies, which is partially true. However, we learned that the OMA addresses component integration in general, while OLE addresses a more limited form of component integration centered on document management. The OMA habitat asserts an "is-a" link between the OLE and OMA problem contexts to capture this subset relationship. Consequently, we deferred experiments relating to the OMA/OLE feature delta until a sufficient range of OMA common facilities for document management have been defined and implemented.

On the other hand, if, rather than component integration, document management architectures were our concern, we might well conduct a comparative study of the OLE component object model and the CORBA object model. Although we did not compare OLE with the OMA in detail, we did use the genealogy and habitat models to develop more detailed feature comparisons between the OMA and PCTE, SunSoft's ToolTalk and remote procedure call (RPC).

OMA/CORBA experiment design

We conducted an extensive comparative anatomy of the OMA and several technologies identified in the OMA genealogy and habitat.

Reference models. In the feature-level reference models we developed, we compared the OMA primarily with PCTE, ToolTalk, and SunRPC. In one instance, we mapped CORBA to a reference model of software environment integration framework technologies. We performed and documented this detailed mapping as part of our evaluation. In a two-step "profiling" process, we first mapped features found in the reference model to CORBA and then mapped CORBA features to the reference model. The first step identifies the features CORBA shares with software development environment integration frameworks, while the second step highlights features of CORBA not found in environment framework technology. Figure 5 shows the first reference model-to-CORBA mapping (with the outer rings corresponding to "greater correlation").

Rough surveys like this can help direct an evaluator's analysis to particular feature classes. For example, the mapping illustrated in Figure 5 led us toward more detailed comparisons of CORBA with communications mechanisms (ToolTalk and RPC) and object management mechanisms (PCTE).
Policy enforcement

Communications

System time versus distributed processing, interactivity versus fault tolerance, and so on. In such cases, knowledge of how and when to delegate features among overlapping technologies can be crucial to meeting application requirements. For example, a real-time or highly secure application might use an ORB to broker communication links between specialized communications hardware and software not managed by the ORB.

**Feature benchmarks.** These helped us understand performance variations among different vendor implementations of the CORBA specification, and also to understand factors in the implementation of object request brokers (ORBs) that influence performance. Additionally, we found it useful to compare the performance characteristics of these CORBA implementations with a commercial RPC implementation to test our assertion that RPC and CORBA are both peers and competitors. Figure 7 illustrates one such feature benchmark, where we compared the performance of three CORBA implementations and Sun/RPC. In other benchmarks, we varied the number of objects and the size of messages, and introduced additional interprocess communication technology such as Sun/Sockets.

Feature benchmarks can be viewed as very simple applications, in terms of structure and algorithms. As such, they are a relatively low-cost means for the evaluator to acquire “hands-on” experience with a technology. We found these simple benchmarks to reveal several dimensions of the CORBA specification, including code portability, inconsistent concepts visible to client writers and object service providers, deficient administrative and security aspects of ORBs, and issues of the robustness of the ORB and ORB-based applications, among others.

**OMA/CORBA experiment evaluation.** The descriptive models and comparative anatomies gave us a good idea of the OMA feature delta: those features

---

Figure 5. CORBA mapping to NIST software environment framework reference model.

Figure 6. Overlapping and disjoint services.

Such surveys are also useful for conducting comparative descriptions of technologies. Consider the feature mapping illustrated in Figure 6, which superimposes a mapping of PCTE onto the CORBA mapping. From this mapping, an analyst can theorize that a system might make use of PCTE services for policy enforcement (for example, security policy) while using CORBA services for interobject communication. Conversely, the analyst might theorize that the object management services overlap sufficiently such that a hybrid PCTE/CORBA system would need to defer all object management functions to either PCTE or CORBA to avoid feature collision.

These conjectures can be neither supported nor refuted, however, by the insufficiently developed reference model and mappings derived from it. This situation suggests experiments, specifically compatibility studies, to determine the kinds of feature interactions that might arise in a hybrid CORBA/PCTE system, or experiments for determining how features might be effectively combined. Determining how OMA/PCTE features can be combined might be important in a large-scale application that has to satisfy competing requirements such as real-time versus distributed processing, interactivity versus fault tolerance, and so on.
that distinguished OMA and CORBA from PCTE, ToolTalk, and RPC (to name a few). In addition, we also had formulated several hypotheses concerning this technology delta's behavior in our selected problem domain (integration of large-scale, off-the-shelf software components). The next step was to define experiment scenarios that would let us apply the OMA feature delta under controlled circumstances and test our hypotheses.

**Model problems.** For our investigations the model problems of interest were derived from the tool and software component integration problem context.

**Experiment overview.** The model problem we describe here concerns architectural mismatch. The problem expressed by this term is that reused, off-the-shelf components embed many integration time and runtime assumptions about their usage contexts, and often the assumptions made by one component are inconsistent with assumptions made by others. One common manifestation of architectural mismatch concerns the locus of control in an application comprising many different large-scale components, that is, which one is in charge?

**Hypotheses.** Architectural mismatch involves a wide range of experiment scenarios and hypotheses, and this is an ongoing area of experimentation for our project—for example, to classify and catalog techniques for removing mismatches. From past experience, we knew that removing them often requires intricate, low-level code. Given the claims that CORBA could be used to encapsulate and integrate legacy applications, we decided to examine CORBA's effect on this code. In particular, we wanted to determine the sensitivity of architectural mismatch solutions to ORB-vendor features. Our hypothesis was that vendor-specific features would have a minor effect on implementation strategies to remove architectural mismatch. Supporting or refuting this hypothesis would let us know to what extent our documented techniques for architectural mismatch removal are vendor-specific.

**Design.** Our experiment design involved the integration of a CORBA object implementation with a graphical user interface into a single executing process. CORBA object implementations typically have event loops that accept requests for services from arbitrary clients; GUI components have event loops that manage display events. For our experiments we used John Ousterhout's TCL/Tk language and toolkit as the GUI component; the object service integrated with the GUI was a simple two-dimensional array that lets clients put and get values at specified locations. We used two different commercially available ORBs. We developed a model solution using one ORB and attempted to "port" this solution to the second ORB. This experiment illustrated the essential characteristics of the problem without introducing extraneous details—the functionality of the final integrated application was trivial while the integration requirements were not trivial and validly represented a potentially challenging integration problem.

**Results.** In brief, the experiment convincingly refuted the hypothesis. We discovered that component integration within a single address space exposed a wide range of ORB vendor-specific idiosyncrasies, and the solutions we selected for one ORB could not be implemented in the other. Although we knew that the OMA doesn't support source code portability for object implementations, the variation between our two model solutions was more dramatic than anticipated. We determined that it would have been possible to implement a model solution that, with minor reprogramming, would work on both ORBs. However, this uniform solution would be considerably more complex than either ORB-specific solution, and we would have only low confidence that the solution would apply to a third ORB with different vendor features.

Regardless of the specific interpretations of our experiment results, we derived much information about the OMA and its exemplar implementations with modest cost and development effort (approximately 200 source lines of C++ for each model solution). This level of effort indicates a well-constructed model problem.

**Demonstrators.** Ultimately, a technology's impact is felt over all application development issues. Experienced software engineering practitioners are well acquainted with technologies that provide leverage early in a software life cycle only to introduce more compelling problems later.
The model problem just described focused on relatively narrow aspects of the OMA. In contrast, demonstrators reveal a technology's broader characteristics when applied to a representa-
tive "complete" problem in an application domain. We developed two OMA demonstrators: (1) integration and wide-area distribution of a collection of legacy software components, which we discuss, and (2) an open, distributed-object-based work flow process definition and enactment framework.

Experiment overview. The experiment involved the OMA's support of flexible integration and wide-area distribution of legacy manufacturing engineering design components. The application domain was manufacturing engineering design. This experiment was conducted collaboratively by the National Institute of Standards and Technology Manufacturing Engineering Laboratory, Sandia National Laboratories, and SEI.

Hypotheses. In contrast to the model problem described above, our concern in this experiment was less on ORB vendor-specific issues and more on how OMA-defined features supported a specific class of integration problems. Our hypothesis was that, as compared to RPC-based approaches, the OMA feature delta would result in an integration framework that would be:

- more abstract and thus easier to maintain, evolve, and standardize;
- more flexible with respect to component location, coordination, and other runtime concerns; and
- more responsive to system-level issues such as data management, persistence, and transactions.

We also were concerned that ORB vendors didn't uniformly implement OMA features beyond CORBA, that is, CORBA products need not be bundled with implementations of Common Object Services or Common Facilities, as shown in Figure 2. Therefore, we further stipulated that OMA services not provided by an ORB vendor could be either partially or fully implemented by application developers with only a modest increase in application development effort.

Design. Our experiment design involved the integration of several Sandia-provided manufacturing engineering design tools. The integration experiment focused on advanced OMA features rather than a more primitive RPC-like use of CORBA. As a result,

- an object model for the engineering design activities performed with the tools was modeled in CORBA IDL;
- OMA services such as persistence and relationships were used for the object model to support long-running, wide-area distributed design sessions;
- the Sandia tools were "wrapped" to fit into the distributed object model rather than wrapped to export tool-specific functionality, and distributed across different sites (at the SEI and NIST); and
- an end-user virtual interface was implemented to make the location and identity of the Sandia tools transparent to end users.

We sketched a paper design of this system using simple RPC primitives to establish a comparative (if hypothetical) baseline. We went so far as to give RPC the "benefit of the doubt" that we used CORBA/IDL as an RPC interface specification language, rather than the more primitive specification languages supported by most existing RPC implementations.

This design addressed the hypothesis as well as the underlying application domain. The total effort devoted to the main design and implementation phase of the demonstrator was approximately six person-months, applied by two software engineers over a period of three months. This was substantial enough to construct a nontrivial demonstrator.

Results. The major elements of the OMA feature delta—object services and an object-oriented interface description—were an excellent foundation for designing and implementing distributed component-based systems; the first part of our hypothesis was largely sustained. However, we discovered that, in practice, developers will require an even richer set of object services than the one we used; it remains to be seen whether vendors will provide implementations of all of these services.

We also convincingly refuted the second part of our hypothesis: We demonstrated that custom implementation of OMA services such as relationship management is impractical, and the use of nonstandard services, such as vendor-specific persistence mechanisms, introduces additional problems, such as coding complexity and nonorthogonality with OMA concepts.

As with all of our experiments, we established practical results beyond the immediate hypotheses. From this experiment we discovered a variety of software architecture-related facets of the OMA. In particular, we discovered the affinity of OMA concepts, when applied to wide-area component integration, to a hybrid repository style and structural style. That is, the OMA both suggested—and was sufficiently flexible to implement—this hybrid style. We demonstrated how the style addressed many technical requirements for flexible, evolvable wide-area component integration; and, further, how
it addressed the sometimes ambiguous, inconsistent, or incompletely specified runtime semantics of OMA specifications.

**Delta study.** The OMA feature delta study shows that OMA exemplifies a new class of technologies believed to significantly affect the design and implementation of distributed systems. Our technology delta framework played an important role in this technology investigation by:

- separating the different evaluation tasks into manageable pieces;
- suggesting an ordering, or methodology, for carrying out different kinds of evaluations; and
- letting the information collected be considered as part of a larger picture that offers a more complete understanding of the technology.

While software technology evaluation is regarded as vital by many organizations, most organizations carry out those evaluations without clearly defining their goals and expectations, and rely heavily on the intuition and experience of the evaluators. We have presented the basis for a systematic approach to software technology evaluation by examining that technology in relation to its peers and predecessors.

Areas for future research include the following:

- Additional rigor in modeling genealogies and habitats. The current semantic-network approach has great flexibility but at the cost of some precision and repeatability. We hope to further define the modeling language in these areas through further application of the framework.
- Improved integration of metrics techniques. The ultimate goal of any evaluation is to define and apply appropriate quantitative techniques that yield objective data on which to base a decision.
- Application to a wider set of technologies. Currently, the technology delta framework has been applied only to component integration technologies. While we expect the concepts to be readily transferable to other domains, we'll have to validate this assertion through its application to candidate technologies.

The technology delta framework evolved from other evaluations, culminating with our study of OMA and CORBA. We plan to apply this framework to evaluate other classes of system integration technologies—for example, scripting languages such as Java <http://java.sun.com> and Python <http://www.python.org>—and for the use of the OMA in domains other than component integration; for example, real-time, highly reliable systems.

---

**ACKNOWLEDGMENTS**

The SEI is sponsored by the US Department of Defense; NIST/MEL sponsored the SEI participation in the demonstrator experiment described in this article.

**REFERENCES**


---

**Alan W. Brown** is a senior member of the technical staff at the SEI, Carnegie Mellon University, and leads the CASE Environments project. Brown was a lecturer in the computer science department at the University of York and a research associate at the University of Newcastle upon Tyne. He has published many papers and several books, including Principle of CASE Tool Integration, Object-Oriented Database and their Application in Software Engineering, and Software Engineering Environment. Brown received a BS in computational science from University of Hull and a PhD in computing science from University of Newcastle upon Tyne.

**Kurt C. Wallnau** is a member of the technical staff at the Software Engineering Institute, Carnegie Mellon University. He is actively involved in several SEI projects in the areas of component integration, software architecture, and CASE, and has several publications in these areas. Prior to joining the SEI, Wallnau was system architect of the US Department of Defense CARS reuse program. Prior to this, he worked in the SEI's software environments project and was a chief programmer for various software-environment-related projects in Army's STAR program. Wallnau received a BS in computer science from Villanova University.