

# Attribute-Based COTS Product Interoperability Assessment

Jesal Bhuta, Barry Boehm  
Center for Systems and Software Engineering  
University of Southern California  
{jesal, boehm}@usc.edu

## Abstract

*Software systems today are frequently composed from prefabricated commercial components that provide complex functionality and engage in complex interactions. Such projects that utilize multiple commercial-off-the-shelf (COTS) products often confront interoperability conflicts resulting in budget and schedule overruns. These conflicts occur because of the incompatible assumptions made by developers of these products. Identification of such conflicts and planning strategies to resolve them is critical for developing such systems under budget and schedule constraints. Unfortunately, acquiring information to perform interoperability analysis is a time-intensive process. Moreover, increase in the number of COTS products available to fulfill similar functionality leads to hundreds of COTS product combinations, further complicating the COTS interoperability assessment landscape. In this paper we present a set of attributes that can be used to define COTS interoperability-specific characteristics. COTS product definitions based on these attributes can be used to perform high-level and automated interoperability assessment to filter out COTS product combinations whose integration will not be feasible within project constraints. In addition to above stated attributes, we present a tool that can be used to assess COTS-based architectures for interoperability conflicts, reducing the overall effort spent in performing interoperability analysis. Our preliminary experience in using the framework indicates an increase in interoperability assessment productivity by 50% and accuracy by 20%.*

## 1. Introduction

Economic imperatives are changing the nature of software development processes to reflect both the opportunities and challenges of using commercial-off-the-shelf (COTS) products. Processes are increasingly moving away from the time-consuming development of custom software from lines of code towards assessment, tailoring, and integration of off-the-shelf (OTS) or other reusable components. Our experiences in developing USC's annual series of rapidly

developed campus electronic services applications has shown an increase in percentages of COTS-Based Applications (CBAs) from 28 percent in 1997 to 70 percent in 2002 [28]. The Standish Group's 2000 survey found similar results (54 percent) in industry [25]. COTS-based systems provide several benefits such as reduced upfront development costs, rapid time to deploy, and reduced maintenance and evolution costs. These economic considerations often entice organizations to piece together their software systems with pre-built components. However these benefits are accompanied by several risk factors such as high maintenance costs, inaccessible source-code and no control over evolution of COTS products [4].

One such risk factor is that of interoperability amongst selected COTS products. The first example of such an interoperability issue was documented by Garlan et al. in [17] when attempting to construct a suite of software architectural modeling tools using a base set of 4 reusable components. Garlan et al. termed this problem architectural mismatch and found that it occurs due to (specific) assumptions that an OTS component makes about the structure of the application in which it is to appear that ultimately do not hold true. The best-known solution to identifying architectural mismatches is prototyping COTS interactions as they would occur in the conceived system. Such an approach is extremely time and effort intensive. Alternately development teams often times manually assess their COTS-based architectures to identify mismatches. Such assessments also take significantly long due to incoherent documentation provided by COTS vendors. This problem is further compounded by the present-day COTS market where there are a multitude of COTS product choices for any given functionality, increasing the number of COTS combinations that would need to be assessed for interoperability.

In this paper we propose a set of 38 attributes that can be used to represent COTS products in order to assess COTS-based architectures for interoperability. An organization can use these attributes to build a COTS definition repository and reuse COTS

interoperability research to perform high level interoperability assessments. These attributes can be used to identify three major interoperability mismatches:

1. Interface (or packaging) mismatches, which occur because of incompatible communication interfaces between two components [24].
2. Internal assumption mismatches, which are caused due to assumptions made by interacting COTS' systems about each other's internal structure [16].
3. Dependency mismatches, which occur when the facilities required by COTS packages used in the system are not being provisioned (e.g. Java-based CRM solution requires Java Runtime Engine).

We have identified these attributes out of the literature and our experiences in developing COTS-based systems. Each attribute has been selected such that they do not expose the internal workings of COTS products while at the same time enable identification of critical interoperability mismatches. This research is primarily focused on technical interoperability issues and does not address non-technical interoperability issues such as human-interface interaction, and economic or strategic.

Rest of the paper is structured as follows. Section 2 describes the definitions and related work. Section 3 defines the proposed COTS attributes. Section 4 describes a prototype tool that utilizes the attributes to perform COTS interoperability assessment. Section 5 presents our experiments with the framework and corresponding results. Finally section 6 rounds out the paper and presents future directions.

## 2. Definitions and Related Work

### 2.1. Definitions

We adopt the SEI COTS-Based System Initiative's definition [10] of a COTS product as a product that is:

- sold, leased, or licensed to the general public;
- offered by a vendor trying to profit from it;
- supported and evolved by the vendor, who retains the intellectual property rights;
- available in multiple identical copies;
- used without source code modification.

For the purpose of this work we include open-source products as part of the COTS domain. This is because although source code for these products is freely available, they are most often used without modification. Moreover there have been several revenue-generating models around open-source [18]. In this paper, we define a component generally as a unit of computation or data store [22]. Components may be as small as a single procedure or as large as an

entire application. Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions [22]. For the purpose of this paper we assume that the source code of the COTS products will not be modified, moreover tailoring [9] of COTS products will not impact any of the definition of the attributes.

### 2.2. Related Work

Researchers have proposed several COTS component selection approaches [4][9][10][11]. Of these, [10][11] are largely geared towards the selection and implementation of COTS based on business and functional criteria. The approach presented by Mancebo et al. in [20] focuses on a COTS-selection process based architectural constraints and does not address the interoperability issue. Ballurio et al. [4] provide a detailed method for assessment of COTS component interoperability. Unfortunately the method requires manual (and usually effort intensive) search of COTS interoperability characteristics. In addition their method recommends filtering COTS first based on functional and non-functional criteria, reducing the COTS interoperability options in the process.

Yakimovich et al. in [26][27] have proposed an incompatibility model that provides a classification of COTS incompatibilities and strategies for their resolution across the system (hardware and software) and environment (development and target) related components. However identification and integration strategies recommended are extremely high-level and require manual analysis for incompatibility identification.

Davis et al. [12][13] present notations for representing architectural interactions, to perform multi-phase pre-integration analysis for component-based systems. The authors define a set of problematic component interactions and 21 components characteristics for identifying them. They further recommend the use of existing [19][23] architectural resolution strategies. Most characteristics however require access to and an understanding of the source-code, which makes this approach complicated to use for COTS systems. Moreover this approach does not identify several interface level attributes, as our own set does. While our approach is similar, it is applicable for components whose source code is either inaccessible or complicated to understand.

Gacek [16] investigates the problem of architectural mismatch during system composition. She presents 14 conceptual features, extending work in [1] that would be used to identify architectural mismatches during system composition. Using these conceptual features

author defines 46 architectural mismatches across six connector types: call, spawn, data connector, shared data, trigger, and shared resource. Our work draws significantly from this research.

Mary Shaw in [24] defines the problem of packaging mismatch. Packaging mismatch occurs when “components make different assumptions about how data is represented, about the nature of system interactions, about specific details in interaction protocols, or about decisions that are usually explicit.” Robert DeLine expands work done by Mary Shaw in [14]. He defines packaging mismatch problem as: “when one or more of a component’s commitments about interaction with other component are not supported in the context of integration.” In the same work he identifies a set of aspects, which define component packaging. DeLine further surveyed a set of mismatch resolution techniques that can be used to resolve such packaging mismatches. Our tool uses several of these mismatch resolution techniques to provide interoperability analysis results.

Mehta et al. [23] propose a taxonomy of software connectors. In the taxonomy authors provide four major service categories addressed by connectors. These include: communication, conversion, coordination and facilitation. They further identify eight primitive connectors classified along a set of dimensions and sub-dimensions unique to each connector type. Our work utilizes these service categories as well as the connector classification for identification of COTS interfaces.

### 3. COTS Representation Attributes

The COTS Representation Attributes are a set of 38 characteristics that define COTS product interoperability characteristics. These attributes have been derived from the literature, as well as our observations in various software integration projects. The two major criteria used for selecting these attributes were:

1. they should be able to capture enough details on major sources of COTS product mismatches (interface, internal assumption and dependency mismatches),
2. they should be defined at high-level so that COTS vendors are comfortable providing relevant information for these attributes

To date, we have surveyed about 40 COTS products of which 30 were open source. For the non-open

source COTS we could identify at least 34 attributes from the publicly accessible information itself. We neglected to include several attributes such as data topology, control structure, and control flow because they were either too detailed and required understanding of internal designs of COTS products for defining them, or could alternately be represented at a higher level by an already included attribute, or did not provide for significant mismatches to warrant us including them.

We have classified the attributes that we selected into four groups. **COTS general attributes** aid in the identification and querying of COTS products. **COTS interface attributes** define the interactions supported by the COTS product. An interaction is defined by the exchange of data or control amongst components. COTS products may have multiple interfaces in which case it will have multiple interface definitions. For example: apache will have one complete interface definition for the web-interface (interaction via http), and another complete definition for server backend interface (interaction via procedure call). When developing the COTS product the developer make certain assumptions about the internal operations of the COTS products. **COTS internal assumption attributes** define such assumptions. For example developers of apache assume that the software will contain a central control unit which will regulate the behavior of the system. **COTS dependency attributes** define the dependencies required by a COTS product. Dependencies of COTS software are products that it requires for successful execution. For example any Java-based system requires the Java Runtime Environment as a platform.

The complete list of attributes is shown in Table 1. In the table attributes (or attribute sets) marked with \* indicate that there may be multiple values for a given attribute (or set) for the given COTS product. For a more detailed definition of these attributes please refer to [6]. Table 1 also includes example definition of Apache Web Server [3] alongside attribute definitions. Please note that multiple versions of COTS products will have multiple definitions. In cases where definitions for different versions of COTS products are exactly the same a single definition can be mapped to multiple versions by adding those version numbers in the version attribute.

**Table 1** COTS Representation Attribute List and Example

COTS General Attributes (4)			Example	
Name	Name of the COTS product		Apache	
Role*	Role of COTS product in the system. E.g. Operating system, Platform, Middleware, ...		Platform	
Type	Product type E.g. third-party component, custom component, legacy component, ...		Third-party component	
Version*	COTS product version		2.0	
COTS Interface Attributes* (14)			<i>Backend Interface</i>	<i>Web Interface</i>
Binding*	Defines how interaction mechanisms are setup, and how the participants of interactions are determined. Values include static binding, compile-time dynamic, run-time dynamic, and topologically dynamic.		Runtime Dynamic	Topologically Dynamic
Communication Language Support*	Communication languages supported by the COTS. E.g. .Net languages for Microsoft Office		C, C++	
Control	Inputs*	Indicates control input techniques supported by the COTS package. E.g. procedure calls, triggers, remote procedure calls, ...	Procedure call, Trigger	
	Outputs*	Indicates control output techniques supported by the COTS package. E.g. procedure calls, triggers, remote procedure calls, ...	Procedure call, Trigger, Spawn	
	Protocols *	Indicates control protocols supported by the COTS package. E.g. ADODB		
Data	Inputs*	Indicates data inputs techniques supported by the COTS package. E.g. procedure calls, shared data, ...	Data access, Procedure call, Trigger	
	Outputs*	Indicates data output techniques supported by the COTS package. E.g. procedure calls, shared data, ...	Data access , Procedure call, Trigger	
	Protocols*	Indicates data communication protocols supported by the COTS package. E.g. HTTP, FTP, ODBC, ...		HTTP
	Format*	Indicates the format in which data is represented for a given interaction. E.g HTML, Javascript, SQL query, ...	N/A	N/A
	Representation*	Indicates the manner in which data is represented during transfer. E.g. ascii, binary, Unicode, ...	Ascii, Unicode, Binary	Ascii, Unicode, Binary
Error Handling	Inputs*	Error inputs (produced by the communicating product) supported by COTS product. E.g. HTTP error codes		
	Outputs*	Error outputs supported by COTS product.	Error Logs	HTTP Error

		E.g. HTTP error codes		Codes
Extensions*		Indicates type of extensions COTS product supports. E.g. Plug-ins	Supports Extensions	None
Packaging*		Indicates how the component is packaged. Values include source code modules, object modules, dynamic libraries, executable programs	Executable Program	Web interface
<b>COTS Internal Assumption Attributes (16)</b>				
Backtracking		Indicates if the COTS product supports backtracking to an earlier state. Values include yes and no	No	
Component Priorities		Indicates if the COTS product supports priorities in execution of tasks, where some tasks have a higher priority over other tasks. Values include yes and no	No	
Concurrency		Indicates the number of concurrent threads that may execute within the COTS product. Values include single-threaded and multi-threaded	Multi-threaded	
Control Unit		Indicates the type of control responsible, which subcomponents are to execute at a given point in time. Values include central control unit, distributed control unit, none	Central	
Distribution		Indicates if the COTS product is mapped to a single node or multiple nodes	Single-node	
Dynamism		Indicates if the COTS product allows for a change in its control topology while it is running. This includes addition and termination of concurrent threads as it executes. Value include static or dynamic	Dynamic	
Encapsulation		Indicates if the COTS product provides users with a well-defined interface to a set of functions in a way that hides internal workings. Values include encapsulated, non-encapsulated	Encapsulated	
Error Handling Mechanisms		Indicates error handling mechanisms supported by COTS product. E.g. redundancy, rollback, roll-forward with compensation, retrying an operation, requesting assistance, notification, none	Notification	
Implementation Language*		Indicates the language used to develop the COTS product. COTS developers may have used multiple languages	C++	
Layering		Indicates that the systems within the COTS product are organized hierarchically, each layer providing a virtual machine to the layer immediately above and serving a client immediately to the layer below. Values	None	

	include control connector layering, data connector layering, control & data connector layering, none	
Preemption	Indicates if COTS product supports interrupting one task and suspending it to run another task. Values include present and absent	Present
Reconfiguration	Indicates if the COTS product can perform reconfiguration online in the event of a failure (or special conditions), or an offline intervention is required to perform reconfiguration. Also indicates of the garbage collection is performed online or off line. Values include online, offline, online with on the fly garbage	Offline
Reentrant	Indicates if the COTS product has multiple simultaneous, interleaved, or nested invocations that will not interfere with each other. Values include yes and no.	Yes
Response Time	Indicates if the COTS product requires that the response for certain events be predictable, bounded or even unbounded. Certain COTS products may contain cycles in which case the product has an unbounded response time. Values include predictable, bounded, unbounded, and cyclic	Bounded
Synchronization	Refers to whether a component blocks when waiting for a response. Values include synchronous and asynchronous	Asynchronous
Triggering Capability	Indicates if the COTS product supports software equivalent of hardware interrupts. Values include yes and no	Yes
<b>COTS Dependency Attributes (4)</b>		
Communication Dependency	Certain COTS products may restrict the pool of components they will interact with, which are required since they provide additional required functionalities	None
Deployment Language	Indicates the language of deployment used by the COTS product	Binary
Execution Language Support	Indicates the support the COTS product may provide for execution of a language (mostly relevant to platform). For example, PHP interpreter supports execution of PHP scripts.	CGI
Underlying Dependency	Indicates dependencies that a COTS product requires for successful execution.	Linux, Unix, Windows, Solaris (OR)

## 4. Interoperability Assessment Tool

To automate the interoperability assessment for COTS-based architectures we have developed a tool at USC. The tool inputs high-level COTS-based deployment architectures and COTS interoperability characteristics defined using the aforementioned attributes (in XML). Every interaction in the

analysis the tool outputs a report which includes the three groups of assessment results – interface, dependency, and internal assumption. This report can then be used by the developers to identify the amount of effort that will be required to integrate the COTS products. In addition to the tool we have also created a PHP-based webpage which enables developers to

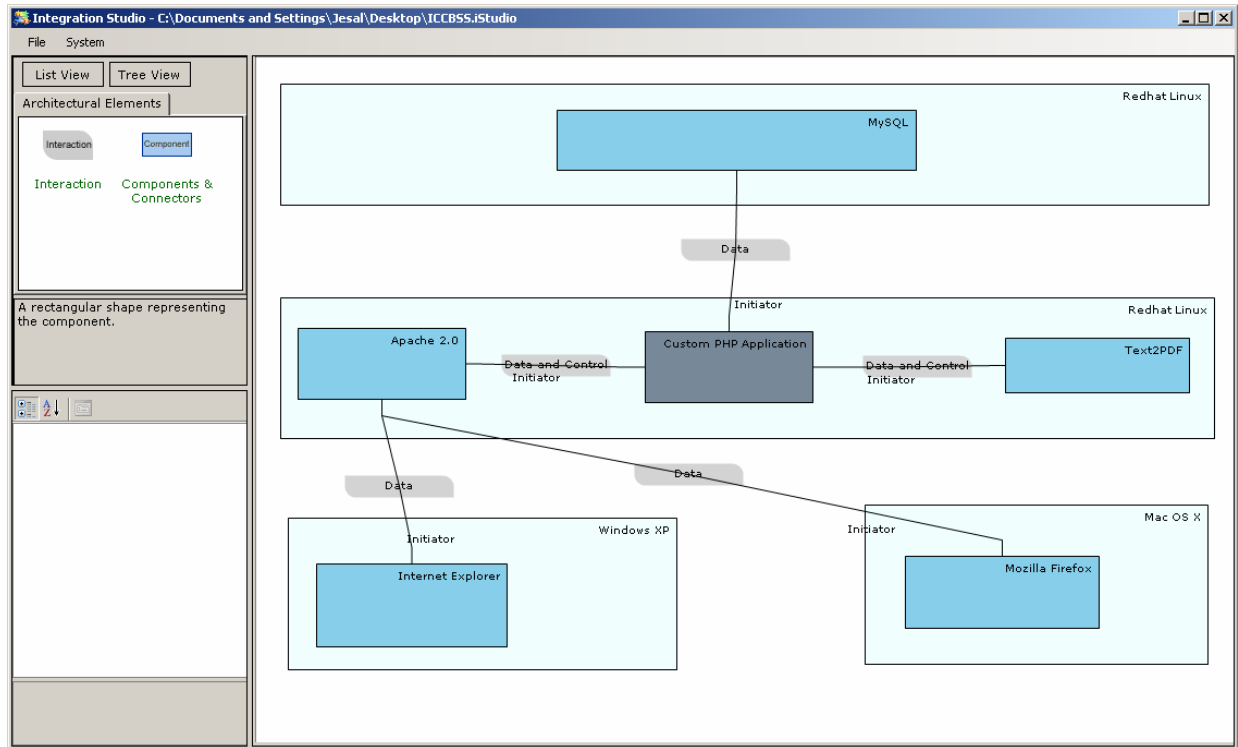


Figure 1 COTS Interoperability Analysis Tool Screenshot

architecture is specified by:

- Type of interaction - control and/or data
- Direction of interaction - unidirectional, bidirectional
- Initiator of interaction - which component initiated the interaction

In addition using the deployment architectures the tool determines the system deployment topology as well as distribution of COTS products. Figure 1 shows the tool interface and illustrates the deployment diagram of a simple 3-tier web application.

The tool utilizes a set of rules for performing interoperability assessment. These include rules to assess for: interface mismatches (and recommend resolution techniques including utilization of third-party connectors), internal assumption mismatches, and dependency mismatches. Upon completion of

create XML-based COTS product definitions and reuse them across multiple assessments in the organization.

## 5. Tool Application and Results

In spring semester of 2006 we conducted an experiment in a graduate software engineering course at USC using our assessment framework. The course focuses on development of software system [8] requested by a real-world client. Over the last few years this course has produced systems for e-services, research (medicine and software), as well as commercial business domains. Graduate students enrolled in the course form teams of about 5 to 6 members to design and implement a software system within a 24-week time period. During this period the project progresses through inception, elaboration, construction, and transition phases [6]. Our

experiment was conducted close to the end of the elaboration phase, when the team proposes a system architecture that would meet the system requirements. We asked 6 teams, whose architectures included at least 3 or more COTS to use our tool on their respective projects and measured results in four areas:

1. Accuracy<sup>1</sup> of interface incompatibilities identified by the framework calculated as  $1 - (\text{number of interface incompatibilities missed by the team} / \text{total number of interface incompatibilities})$ . Interface assessment results produced by our framework were verified later through a survey when the teams actually integrated the COTS products. Results in this area evaluate the completeness and correctness of our interface assessment rules.
2. Accuracy<sup>1</sup> of dependencies identified by the framework calculated as  $1 - (\text{number of dependencies missed by the team} / \text{total number of dependencies})$ . Dependency assessment results produced by our framework were also later verified through a survey after the project was implemented. These results evaluate the completeness and correctness of our interface dependency rules.
3. Effort spent in assessing the architectures using the framework opposed to the effort spent in assessing the architectures manually by an equivalent team. These results demonstrate the efficiency of using our framework to perform interoperability assessment as opposed to performing a manual assessment.
4. Effort spent in performing the actual integration after using the framework as opposed to effort spent by an equivalent team. Results here validate the overall utility of our framework

Equivalent teams were chosen from past CSCI 577 projects such that they had similar COTS products, similar architectures, and whose team-members had similar years of experience in project development.

Upon performing independent T-test [15] for four cases above we recorded the results shown in Table 2. Our results indicate that the framework increases dependency assessment accuracy and interface assessment accuracy by more than 20% and reduces both assessment effort and integration effort by approximately 50%. These results are significant at the  $\alpha = 5\%$  level.

<sup>1</sup> Inaccuracies include both missing incompatibility identifications and false positives (compatible interfaces identified as incompatible).

**Table 2** Empirical Assessment Results when applying Interoperability Assessment Tool

Groups	Mean	Std. Dev.	P-Value
<b>Interface Assessment Accuracy</b>			
Before using the tool	76.9%	14.4	0.0029
After using the tool	100%	0	
<b>Dependency Assessment Accuracy</b>			
Before using the tool	79.3%	17.9	0.017
After using the tool	100%	0	
<b>Effort spent in performing architecture assessment</b>			
Projects using the tool	1.53 hrs	1.71	0.053
Equivalent projects	5 hrs	3.46	
<b>Effort spent when integrating the COTS products</b>			
Projects using the tool	9.5 hrs	2.17	0.0003
Equivalent projects	18.2 hrs	3.37	

The tool's perfect detection record in this experiment indicates that it has a strong "sweet spot" in the area of smaller e-services applications with relatively straightforward COTS components, but with enough complexity that less COTS-experienced software engineers are unlikely to succeed fully in interoperability assessment. We plan to conduct further tool evaluations on larger projects with more complex COTS products.

## 6. Conclusion and Future Work

This paper presents a set of COTS representation attributes that enables interoperability assessment of architectures using COTS product combinations early in the software development life-cycle. Using our attributes and tool does not eliminate detailed testing and prototyping for evaluating COTS interoperability, however it does provide an analysis of interface compatibilities, dependencies, recommends connectors to be used or glue-code required, all of which could be tested for during detailed prototyping. Moreover, since the tool is automated it enables evaluation of large number of architectures and COTS combinations, increasing the trade-off space for COTS component and connector selection. We are in the process of developing a fully functional tool suite which will include a higher level of automation. We are planning experiments and evaluations to gather empirical data to further test the utility of the attributes and tool. In addition, we are collaborating with researchers identifying similar

attributes to assess architectures for quality of service (QoS) parameters. One such QoS extension that is being incorporated in our tool is that on voluminous data intensive interactions [21]. It is important to note that these attributes must be periodically updated based on prevailing COTS characteristics.

## 10. References

- [1] Abd-Allah A., "Composing Heterogeneous Software Architectures" PhD dissertation, University of Southern California, 1996.
- [2] Albert C., Brownsword L., "Evolutionary Process for Integrating COTS-Based Systems (EPIC)," SEI Technical Report CMU/SEI-2002-TR-005.
- [3] Apache Web Server, <http://www.apache.org/>.
- [4] Ballurio K., Scalzo B., Rose L., "Risk Reduction in COTS Software Selection with BASIS," Proceedings of First International Conference, ICCBSS 2002 Orlando, Florida, February 2003.
- [5] Basili V., Boehm B., "COTS-Based Systems Top 10 List," IEEE Computer May 2001.
- [6] Bhuta J., "A Framework for Intelligent Assessment and Resolution of Commercial Off-The-Shelf (COTS) Product Incompatibilities," USC, Tech. Report USC-CSE-2006-608, 2006.
- [7] Boehm B., "Anchoring the Software Process," IEEE Software, July 1996.
- [8] Boehm B., Egyed A., Port D., Shah A., Kwan J., Madachy R., "A Stakeholder Win-Win Approach to Software Engineering Education," Annals of Software Engineering Volume 6 Issue 1-4, 1998.
- [9] Boehm B., Port D., Yang Y., Bhuta J., Abts C., "Composable Process Elements for Developing COTS-Based Applications", 2003 ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2003).
- [10] Brownsword L., Obnerndorf P., Sledge C., "Developing new Processes for COTS-Based Systems," IEEE Software Volume 17, Issue 4 July/August 2000.
- [11] Comella-Dorda S., Dean J., Morris E., Oberndorf P., "A Process for COTS Software Product Evaluation," Proceedings of First International Conference, ICCBSS 2002 Orlando, Florida, February 4-6, 2003.
- [12] Davis L., Gamble R., Payton J., Jónsdóttir G., Underwood D., "A Notation for Problematic Architectural Interactions," 3rd joint meeting of the European Software Engineering Conference, and ACM SIGSOFT's Symposium on the Foundations of Software Engineering, Vienna, Austria, 2001.
- [13] Davis L., Gamble R., Payton J., "The Impact of Component Architectures on Interoperability," Journal of Systems and Software (2002).
- [14] DeLine R., "A Catalog of Techniques for Resolving Packaging Mismatch," Proceedings of the Symposium on Software Reuse 1999.
- [15] Dietterich T., "Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms," Neural Computation, vol. 10, 1998.
- [16] Gacek C., "Detecting Architectural Mismatches During System Composition," PhD dissertation, University of Southern California, 1998.
- [17] Garlan D., Allen R., Ockerbloom J., "Architectural Mismatch or Why it's hard to build systems out of existing parts," International Conference on Software Engineering 1995.
- [18] Hecker F., "Setting up shop: The business of open-source software," IEEE Software, vol. 16, 1999.
- [19] Keshav R., Gamble R., "Towards a Taxonomy of Architecture Integration Strategies," 3rd International Software Architecture Workshop, 1-2, Nov 1998.
- [20] Mancebo E., Andrew A., "A Strategy for Selecting Multiple Components," Proceedings of ACM Symposium on Applied Computing, 2005.
- [21] Mattmann C., "Software Connectors for Highly Distributed and Voluminous Data-intensive Systems," In Proc. ASE, Tokyo, Japan, 2006.
- [22] Medvidovic N., Taylor R., "A Classification and Comparison Framework for Software Architecture Description Languages," IEEE Transactions on Software Engineering, Volume 26, Issue 1, January 2000.
- [23] Mehta N., Medvidovic N., Phadke S., "Towards a Taxonomy of Software Connectors," Proceedings of 22nd International Conference on Software Engineering (ICSE '00) 2000.
- [24] Shaw M., "Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging," Proceedings of the Symposium on Software Reuse (SSR '95), 17th International Conference on Software Engineering (ICSE '95) 1995.
- [25] Standish Group, Extreme CHAOS, tech. report, 2001, <http://www.standishgroup.com>.
- [26] Yakimovich D., Bieman J., Basili V., "Software Architecture Classification for Estimating the Cost of COTS Integration," 21st International Conference on Software Engineering, 1999.
- [27] Yakimovich D., Travassos G., Basili V., "A Classification of Software Component Incompatibilities for COTS Integration," 24th Software Engineering Workshop, December 1999.
- [28] Yang Y., Bhuta J., Boehm B., Port D., "Value-Based Processes for COTS-Based Applications," IEEE Software special issue on COTS Integration, July/August 2005.