Defining Change Management Properties for Component Interoperability Assessment

T. Gamble R. Gamble L. Davis
Department of Mathematical and Computer Sciences
The University of Tulsa
600 South College Avenue
Tulsa, OK 74104 USA
+1 918 631 2988
{todd.gamble, gamble, davis}@utulsa.edu

ABSTRACT

In this paper, we leverage software architecture analysis techniques to codify change management properties and to examine their affect on the functional integration of software components. Software architecture provides a means to predict and analyze the potential for interoperability problems among interacting components. However, the properties addressed in traditional software architecture analysis are limited to the functions for a component's exchange of control and data. We define architectural abstractions for change management that can be used to extend integration analysis. We exemplify how these properties contribute to the architectural conflicts between interacting components, as well as influence the conflict resolution. The goal is to extend component-based systems analysis to include change management concerns so that clearer designs of dynamic, distributed systems emerge.

1. INTRODUCTION

Component-based systems are now commonplace in every industry. The prevalence of Commercial-off-the-Shelf (COTS) software, increased use of middleware, corporate mergers, evolving markets, and higher customer service expectations drive the need for more dynamic software systems. New computing environments, such as the Grid [13] and mobile computing [7, 29], increase the likelihood that distributed component interactions will change over time. To compound the problem, security concerns, change-management issues, and the lack of financial and human resources to recode existing legacy systems increase the complexity of implementing reliable integrations.

Software architecture has emerged as a reliable technique for expressing system design [33]. This is particularly true with respect to integrated applications, in which multiple, distinct, and distributed component software systems must interact. Current research in software architecture isolates problems with system integration by segregating concerns into single-issue strategies, such as describing the architecture and how it changes [24] or describing security protocol interaction [35]. This problem isolation has effectively resulted in the maturation of the field to describe properties across a single set of related concerns and to manipulate those descriptions toward the selected goal.

There is high demand for approaches that encompass a wider range of properties and account for the changes that occur in an integrated system. Thus, change management properties are needed to address dynamic insertion, deletion, and modification to any part of a component-based application. Yet, it is a difficult task to uniformly analyze such a broad range of component-based system design concerns. Furthermore, ensuring these considerations are addressed throughout the life of the application presents an even greater challenge.

Change management software, when part of a component-based system, is responsible for monitoring and controlling (to some extent) the execution of the component. Because the goal is to overload existing component interoperability conflict analysis techniques, the expression of change management concerns should compliment those modeled for software architecture. The challenge is to bridge the gap between abstraction levels, viewpoints, and the segregated concerns change management brings to integrated system design. Therefore, we must consistently and cohesively represent the properties of software architecture and change management concerns of a component, as well as codify how these interrelated but disparate properties should be governed in a manner that is controllable, yet flexible.

In this paper, we examine change management concerns and present their preliminary expression as architecture abstractions. We use these properties and the meaningful information they yield for component-based systems to expand architecture integration assessment toward a holistic approach. Furthermore, we point to how solutions to architecture interoperability conflicts can be reused or overloaded to accommodate change manage-

ment-induced conflicts when the related properties are considered as part of the overall integration analysis.

The connotations of component-based systems, change management, and integration analysis terminology are not universal. Below, we define some of the main terms used in the paper.

- **Component** – A stand-alone system, probably encapsulated, that has one or more exposed interfaces (e.g., COTS, GOTS, and in-house systems) and participates as an actor in a component-based system (or system-of-systems).
- **Change Management** – Properties that reflect the configuration, versioning, deployment, and monitoring expectations of a component and its environment.
- **Architecture Abstractions** – Properties that express the expectations a component has with respect to data and control exchange, along with other processing qualities.
- **Interoperability** – The transparent exchange of data and control between one or more components through the use of middleware.
- **Integration** – The mechanism by which multiple components, both homogeneous and heterogeneous, interoperate within a larger application environment.

2. RELATED RESEARCH

In this section, we review concepts that are germane to change management and software architecture as they relate to interoperability analysis.

The software architecture of a system is the layout of its computational modules, the means by which they interact (connectors), and the structural constraints on their interaction [33]. Using this abstraction, it is possible to focus on conceptual issues without implementation and deployment details. Software architectures are often defined as architectural patterns or styles (e.g., pipe/filter, layered, main/subroutine) [8, 15, 27, 30]. The software architecture of a component (as an independent software system) embodies properties that pertain to a component's potential to deliver clear and early warnings of interoperability problems. Properties of architectural styles include those that describe the various types of computational elements and connectors, data issues, control issues, and control/data interaction [3, 5, 6, 16, 17, 31]. The internal details of data structures, function calls, protocols, etc. are not required for quick assessment: they may be addressed once early conflicts are confirmed.

Architecture mismatch points to properties of architecture styles that denote the underlying reasons for interoperability problems among seemingly "open" software components [16]. Methodologies for analyzing mismatch exist both within and between styles [1]. Data representation, data and control transfer, transfer protocol, state persistence, state scope, failure, and connection establishment are all considered.

The evolutionary aspects of components can also be captured using software architecture. One goal is to find an architecture to which other architectures can transition easily [33]. Though architecture migration is a plausible solution, it is not always feasible for all systems, especially COTS products, where many properties are hidden. In a similar vein, researchers examine constraints on reconfiguring architectures to assess their response to evolution [38]. This illustrates that certain properties of components and connectors lessen the impact of change [28]. Certain ADLs support evolution through topology [28], optionality [40], and variability [32].

There are many approaches to interoperability analysis. Some approaches are based on the different viewpoints that come from system specific integration concerns. Barrett et al. [6] focuses on only the event-based architecture style. Other viewpoints, such as Gruhn et al. [19] and Hofmeister's [34] involve case studies of integration problems and solutions. The Architect's Automated Assistant (AAA) system is based on examining style-based differences, focusing on the description of conventional architectural styles such as event-based, main-subroutine, distributed processes, and pipe and filter [1, 2, 14]. DeLine [12] attempts to resolve package mismatch by describing various aspects of each component to be integrated, e.g., data representation and data/control transfer. Yakimovich et al. [44] combines both high- and low-level analysis. This classification scheme is based on architectural assumptions that can cause mismatches due to the nature of the components and connectors, the global architectural structure, and the construction process for application.

Change management encompasses issues regarding configuration management and component deployment [36, 37, 42, 43]. The research that lends the most credence to our approach uses system modeling to consolidate software architecture, configuration management, and configurable distributed systems [39]. In this work, van der Hoek, et al., outline the benefits of combining these concerns to reduce modeling effort and architecture erosion, to provide higher levels of abstraction for configuration management, and to improve reuse and version control. System models for the software architecture, configuration management, and configurable distributed system disciplines are examined and their contribution assessed over a set of particular capabilities chosen from all three. Their experimentation with adding functionality to existing system models to compensate for absent capabilities shows that "cross-fertilization" of these disciplines is possible.

Application development becomes extremely difficult when applications must be dynamically composed while in active use. Dynamic composition methods warrant a

certain level of system correctness during and after insertion of a new system element. One approach is to model the behavior of the system to manage change and to ensure consistency and completeness [4, 18, 25]. Generic system models are also being formulated that leverage change management concepts such as revisions, variants, and configurations, with architectural concepts like components, connectors, subtypes and styles [20].

Our previous approach to interoperability analysis starts with architectural properties of the participating components and application requirements [10]. We perform an assessment of component properties. Table 1 defines a sample set of these properties. A more extensive set can be found in [9, 11, 22]. The characteristic name appears in column one. Definitions and values are in columns two and three, respectively. Through methodological assessment, we detect potential conflicts, consolidating them across common influences. Integration elements [23] representing translation, control and extension are used to design the integration architecture that bridges the gaps in component interaction.

A major challenge of architecture description and analysis is to allow the component-based system architecture to be modifiable in a dynamic manner. Dynamic architectures vary in their degree of change. There are many factors that can be used to discern the dynamic quality of a system. We focus on integration and the impact of change on constructing and manipulating individual components within the application. The dynamic qualities of versioning, deployment, and monitoring impact how an integration architecture is designed. Different kinds of connections are needed between components depending on the type of communication, the extent to which change is imminent, and the amount of mismatch detected between the components [26].

3. CHANGE MANAGEMENT'S ROLE

In this section, we discuss change management and the qualities of the properties we seek to establish at the architecture level of abstraction. Change management properties dictate and constrain the dynamic nature of the system, i.e., how the system is monitored and controlled as well as how alterations (both dynamic and static) within the system are handled [41].

3.1 Qualities of Architecture Abstraction

In order to express the relevant change management properties at the appropriate level of abstraction, we first must provide a baseline for evaluating the properties under consideration. In this section, we identify what makes a property appropriate for inclusion in architecture assessment.

The property is qualified, i.e., it has known values. Having values means the property can be compared across components or between the component and its environment. Even binary values allow developers to ascertain discrepancies (e.g., whether a component is versioned is a binary property).

Table 1. Select Architecture Characteristics

Characteristics	Definition	Values
Control Structure	The structure that governs the execution in the system	Single-Thread, Multi-Thread, Concurrent
Data Storage Method	How data is stored within a system	Local, Global, Distributed
Identity of Components	Awareness of other components in the system	Aware, Unaware
Supported Control Transfer	The method supported for control transfer	Explicit, Implicit, None
Supported Data Transfer	The method supported for data transfer	Explicit, Shared, Implicit (discreet/continuous), None

Table 2. Change Management Properties

Property	Definition	Values
Monitoring data transfer	How the component reports monitoring information	None, Push (sends), Pull (provides API)
Data collection/forwarding	How the component reports monitored information	Collect-and-forward, Batch-and-forward
Version	How the component is versioned	Yes, No
Configuration Mechanism	How the component receives its configuration commands/info	Command line, File, IPC, DB
Reconfiguration State	What state the component needs to be in for it to be reconfigured	Restart, Quiescent, Runtime
Configuration Topology	The relationships between components that affect the required order of change	Linear, Hierarchical, Concurrent, Etc.
Configuration Access	The comparison between the command/control interface and the functional interface	In-band, Out-of-band

Table 3. Sample Properties for Enabling a Component in an Environment

Category	Characteristics	Environment Values	Component Values
Architecture	Supported Control and Data Transfer	Implicit Event Bus	Explicit
Change Management	Monitoring Data Collection	Batch and forward to a central console	Collect and forward discreet, unbuffered events

The property denotes component expectation or policy. A component's expected behavior often symbolizes what it anticipates from its external interaction. Thus, analysis can be performed to determine if there are conflicting expectations. Policies dictate what and how a service is guaranteed. They cannot be violated by inserting the component into an integrated system, and therefore, must be taken into account during the component-based system design. For example, a complex system consisting of many, inter-related components may require a specific ordering to be followed when reconfiguring the component parts of the overall system. This precedence ordering represents a policy. The expectation is that there are global properties of the composed system that must be maintained by enforcing the policy.

The property is expressed by an exposed interface. A property that is exposed to the external environment is relevant to its interoperability with other components. It is only through interaction via an interface that a conflict occurs. Thus, the problem may surface at the interface level or via a collaboration of components and the environment. For example, if we require monitoring of a collection of components that form a complete application system, the monitoring information must be consistent not only in type and format but also in currency. Each component must report monitoring information that is temporally consistent with its peer components in the collaboration that forms the application. The data collection and forwarding mechanism for each component presents information to the CM environment via an interface. The interfaces for reporting monitored data must be consistent across the composed application.

The property is applicable to most components, including COTS components. COTS products in the form of components, middleware, and change management software are widely available and affordable. They are often closed and provide minimum information, making a complete characterization difficult. An example of this type of property deals with the state the component must reach before a configuration modification can be made. Some components must be fully stopped; others may only require quiescence. Components exhibiting the most robustness will allow reconfiguration while running.

Table 2 shows some common change management properties that have one or more of the above qualities. The first column names the property, followed by its definition and potential values in columns two and three.

3.2 Using Change Management Properties

Most change management systems are software systems in their own right with implementation mechanisms and functionality that may be described as software architectures. While devising the mechanisms to facilitate change management is important, our focus is on the impact that this property type has on functional integration as assessed at the architecture level. Thus, an objective of our research is to address what happens when a configuration change is made, i.e., how the functional architecture

is altered as a result. This includes describing how the desired change is *enacted within the component* and how it will impact other communicating components.

If the properties uncovered are properly abstracted and relevant to interoperability, they can then be used during interoperability assessment. There are two main stages to this assessment: (1) examining the relationship between the component and the environment or framework in which it will be inserted, and (2) examining the relationship among all the components that comprise the integrated application.

3.2.1 Aligning a Component with the Environment

In the first stage, we have to align the global environment constraints with the local component information. Thus, we examine properties with respect to a single component's expectations.

We use a brief example to illustrate the basics of our approach and how this comparison is accomplished. Given that a component awaits insertion into an environment, we analyze select properties to determine the design change or integration solution needed in order to adapt it to the environment. Table 3 expresses a characteristic/value pair for a single property in the communication and change management categories of the environment and the component.

Our architecture interoperability assessment techniques include examining the interplay between the environment and components [11, 21]. For instance, Table 3 indicates an obvious communication mismatch. The environment expects components to be accessed via an implicit event bus. A component that uses explicit invocation will not be able to rendezvous with other components as expected. This inhibited rendezvous conflict means that there may be handshaking problems because the components involved in the exchange have different control transfer assumptions [1, 14], such as different calling structures, independent and non-terminating execution that cannot be pre-empted, or several seats of control.

Similarly, we can assess change management to show a discrepancy in how data collection and forwarding is managed by the environment and performed by the component. Monitoring data is processed in batches versus collecting and immediately forwarding monitoring events. This can cause a data inconsistency conflict. A data transfer, if inconsistently communicated, can permute and corrupt the data content of the communication [1, 14, 27].

A controller is typically used to resolve the architecture conflict by determining where and how data should be sent to maintain consistency. Its design is reusable in the context of the change management properties evaluated. In fact, we can define a controller solution [23] that resolves both problems. For the explicit/implicit conversion, the controller monitors the bus for the appropriate message signifying an implicit invocation and then makes explicit calls to the component API to invoke the appropriate functionality. For the change management problem of data collection and forwarding, the controller implements a buffer to capture the events and put them in batches. It also implements a scheduler to determine when the batched set of data should be sent to the console monitoring change management.

The design of the controller is incorporated into the local integration solution that is associated with the component of concern to minimize cascading changes brought about by dynamic component assertions throughout the environment. Thus, when the characteristics that affect the integration design change, that change can be reflected in the local integration solution.

3.2.2 Component Properties and other Components

The next stage of architecture integration assessment addresses problematic behavior interaction issues within the constraints of a multi-component application. In other words, once a component is able to communicate within the established environment, analysis must consider the interaction of this component with other select components that work together toward some application goal. Interoperability conflicts can surface due to both component interactions and their compliance with application requirements that are separate from the environment properties. To find these conflicts and design their solution as part of the integrated system design, property comparisons must include the interplay between architecture expectations and change management properties.

Figure 1 depicts the scenario of two components, A and B, communicating in a component-based system with defined requirements. Table 4 organizes properties into their categories. (Note that all values do not have to be assigned for meaningful assessment to occur.)

We use this example to show three key issues. First, that change management concerns have a significant role in component-based system design analysis. A second issue is that application properties (such as those on the oval line in Figure 1) constrain how change management is performed. Finally, conflict resolution strategies can be reused and overloaded, minimizing the redundancy and complexity associated with separate and distinct solutions that can cause added problems over time.

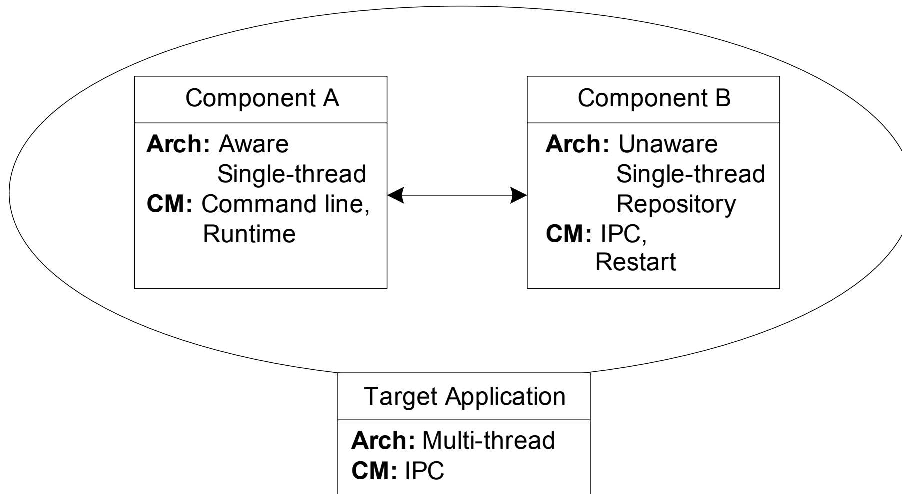

Figure 1. Component Assessment in a Target Application

Table 4: Sample Properties for a Component-Based System

Category	Name	Application Values	Component Values	
			A	B
Architecture	Identity of Components		Aware	Unaware
	Control structure	Multi-thread	Single-thread	Single-thread
	Data storage method			Repository
Change Management	Configuration Mechanism	IPC	Command line	IPC
	Reconfiguration State		Runtime	Restart

Returning to Table 4, the application requires multi-threading where two single-threaded components, A and B, interact. In this case, it is desirable to perform simultaneous updates of A and B (assuring consistency across the application) when they change. The configuration mechanism for the entire target application is specified via an inter-process communication (IPC) interface. B already has such an interface. However, A is configured via a command line interface (CLI), indicating major change-management conflicts between the application and component characteristics. This conflict can be resolved by introducing a translator that allows all incoming configuration commands to the application to be via IPC, adapting those communications to B's CLI.

We assume that A is a client of B. While A is capable of continuous operation even when being reconfigured, B must be restarted for configuration changes. During a system reconfiguration, both A and B must be changed. However A must also discontinue (or take into account) that B will be unavailable for some amount of time. During this restart time, A must compensate for B's denial of access requests. Therefore, A's architecture must be augmented with error handling logic to deal with system inconsistency during B's restart.

This requires the introduction of a controller that mimics responses from B to A. This controller logic can be shared with one needed to resolve the conflict in communication identity issues. In this case, a controller is introduced in order to *make it appear* that B is aware of A and can respond to A directly. Therefore, the functionality can be overloaded to resolve both problems within the same integration solution.

4. CONCLUSIONS

The implementation of dynamic, heterogeneous, component-based systems can be error-prone unless proper integration analysis and design is available to guide development. Complex interoperability problems that arise in this type of system deployment must be resolved holistically in order to maintain and evolve the system. Incorporating change management properties into interoperability assessment should not add a level of complexity to the analysis. Rather, it should increase the prospects of designing a minimal, consistent, and complete solution to

resolve conflicts. We move toward this goal by raising change management concerns to the level of abstract properties that can be viewed using the foundation in place for software architecture assessment. Moreover, we show that the assessment allows for reuse and overloading of the functionality needed to form integration solutions.

Future research in this area seeks to prioritize the assessed components, properties, and values to streamline integration solution design. Furthermore, the integration solution itself needs assigned properties that reflect change management concerns such as configuration topology and versioning (Table 2) and their influence on design parameters.

5. ACKNOWLEDGEMENTS

This material is based upon work supported in part by AFOSR (F49620-98-1-0217) and NSF (CCR-9988320). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the US government. The government has certain rights to this material.

6. REFERENCES

[1] Abd-Allah, A. Composing Heterogeneous Software Architectures. Ph. D. Dissertation, Computer Science, University of Southern California, 1996.

[2] Abd-Allah, A., Boehm, B. Models for Composing Heterogeneous Software Architectures. USC-CSE-96-505. University of Southern California, 1996.

[3] Abowd, G., Allen, R., Garlan, D. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodologies* (1995), **4**(4): 319-364.

[4] Allen, R., Douence, R., Garlan, D. Specifying Dynamism in Software Architectures. In, *Foundations of Component-Based Systems Workshop*, (1997).

[5] Allen, R., Garlan, D. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodologies* (1997), **6**(3): 213-249.

[6] Barret, D., Clarke, L., Tarr, P., Wise, A. An Event-Based Software Integration Framework. 1995.

[7] Bauer, M., Bruegge, B., Klinker, G., MacWilliams, A., Reicher, T., Sandor, C., Wagner , M. An Architecture Concept for Ubiquitous Computing Aware Wearable Computers. In, *International Workshop on Smart Appliances and Wearable Computing*, (2002). Vienna, Austria.

[8] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*: John Wiley & Sons. 1996.

[9] Davis, L., Gamble, R. The Impact of Component Architectures on Interoperability. *Journal of Systems and Software* (2002).

[10] Davis, L., Gamble, R., Payton, J., Jónsdóttir, G., Underwood, D. A Notation for Problematic Architecture Interactions. In, *The third joint meeting of the European Software Engineering Conference, and ACM SIGSOFT's Symposium on the Foundations of Software Engineering*, (2001). Vienna, Austria.

[11] Davis, L., Payton, J., Gamble, R. How System Architectures Impede Interoperability. In, *2nd International Workshop On Software and Performance*, (2000).

[12] DeLine, R. Techniques to Resolve Packaging Mismatch, In, *21st International Conference on Software Engineering* (1999). Los Angles, CA.

[13] Foster, I., Kesselman, C. *The Grid: Blueprint for a New Computing Infrastructure*. 1 ed. San Francisco: Morgan Kaufmann Publishers. 1998.

[14] Gacek, C. Detecting Architectural Mismatches During Systems Composition. Usc/cse-97-tr-506. 1997.

[15] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns Elements of Reusable Object-Oriented Software*: Addison-Wesley. 1995.

[16] Garlan, D., Allen, A., Ockerbloom, J. Architectural Mismatch, or Why it is Hard to Build Systems out of Existing Parts. In, *ICSE*, (1995). Seattle, WA.

[17] Garlan, D., Monroe, R., Wile, D. ACME: An Architectural Description Language. In, *Cascon*, (1997).

[18] Goedicke, M., and Meyer, T. Dynamic Semantics Negotiation in Distributed and Evolving CORBA Systems: Towards Semantic-Directed System Configuration. In, *International Conference on Configurable Distributed Systems*, (1998).

[19] Gruhn, V., Wellen, U. Integration of Heterogeneous Software Architectures- An Experience Report, *Software Architecture*, P. Donohoe, Editor. Boston, MA Kluwer Academic Publishers. (1999).

[20] Hoek, A., Mikic-Rakic, M., Roshandel, R., Medvidovic. Taming Architectural Evolution. In, *ESEC/FSE*, (2001). Vienna, Austria.

[21] Jónsdóttir, G., Flagg, D., Davis, L., Gamble, R. Integrating Components Incrementally. In, *International Conference on Software Engineering and Applications (IASTED)*, (2002). Cambridge,

MA: Department of Mathematical and Computer Sciences, The University of Tulsa.

[22] Kelkar, A., Gamble, R. Understanding the Architectural Characteristics behind Middleware Choices. In, *1st International Conference in Information Reuse and Integration*, (1999).

[23] Keshav, R., Gamble, R. Towards a Taxonomy of Architecture Integration Strategies. In, *3rd International Software Architecture Workshop*, (1998).

[24] Khare, R., Guntersdorfer, M., Oreizy, P., Medvidovic, N., Taylor, R. xADL: Enabling Architecture-Centric Tool Integration with XML. In, *Proceedings of the 34th Hawaii International Conference on System Sciences*, (2001).

[25] Lim, A. Abstraction and Composition Techniques for Reconfiguration of Large-Scale Complex Applications. In, *IEEE International Conference on Configurable Distributed Systems*, (1996). Annapolis, Maryland.

[26] Mikic-Rakic, M., Medvidovic, N. Middleware for Software Architecture-Based Development in Distributed, Mobile, and Resource-Constrained Environments. USC-CSE-2002-508. University of Southern California, 2002.

[27] Mularz, D. Pattern-Based Integration Architectures. In, *PLoP*, (1994).

[28] Oreizy, P., Medvidovic, N., Taylor, R. Architecture-Based Runtime Software Evolution. In, *20th Int'l Conf. on Software Engineering*, (1998). Kyoto, Japan.

[29] Roman, G.-C., Julien, C., Huang, Q., ", P.o.t., . Formal Specification and Design of Mobile Systems. In, *7th International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, (2002). Nice, France.

[30] Shaw, M. Some Patterns for Software Architectures, (1995).

[31] Shaw, M., Clements, P. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In, *1st International Computer Software and Applications Conference*, (1997). Washington, D.C.

[32] Shaw, M., Deline, R., Klien, D., Ross, T., Young, D., Zelesnik, G. Abstraction for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering* (1995), **21**(4).

[33] Shaw, M., Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall. 1996.

[34] Soni, D., Nord, R., Hofmeister, C. Software Architecture in Industrial Applications. In, *International Conference on Software Engineering*, (1995). Seattle, WA: ACM Press.

[35] Spitznagel, B., Garlan, D. A Compositional Approach for Constructing Connectors. In, *The Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, (2001). Amsterdam , The Netherlands.

[36] Tosic, V., Mennie, D., Pagurek, B. Software Configuration Management Related to Management of Distributed Systems and Services and Advanced Service Creation. In, *Tenth International Workshop on Software Configuration Management (SMC-10)*, (2001). Toronto, Canada.

[37] van der Hoek, A. Integrating Configuration Management and Software Deployment. In, *Working Conference on Complex and Dynamic Systems Architecture (CDSA 2001)*, (2001). Brisbane, Australia.

[38] van der Hoek, A., Heimbigner, D., Wolf, A. Capturing Architectural Configurability: Variants, Options, and Evolution. CU-CS-895-99. University of Colorado, 1999.

[39] van der Hoek, A., Heimbigner, D., Wolf, A. Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage a Trois. University of Colorado, 1998.

[40] van Ommering, R., van der Linden, F., Kramer, J., Magee, J. *The Koala Component Model For Consumer Electronics Software*, in *IEEE Computer*. 2000. p. 78-85.

[41] Waheed, A., Rover, D.T., Hollingsworth, J.K. Modeling and Evaluating Design Alternatives for an On-Line Instrumentation System: A Case Study. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* (1998), **24**(6).

[42] Weber, D.W. Requirements for an SCM Architecture to Enable Component-Based Development. In, *Tenth International Workshop on Software Configuration Management (SCM-10)*, (2001). Toronto, Canada.

[43] Westfechtel, B., Conradi, R. Software Architecture and Software Configuration Management. In, *10th International Workshop on Software Configuration Management (SCM 10)*, (2001). Toronto, Canada.

[44] Yakimovich, D., Travassos, G.H., Basili, V. A Classification of Software Components Incompatibilities for COTS Integration. In, *Software Engineering Laboratory Workshop*, (1999).