Architectural Style Requirements for Self-Healing Systems

Marija Mikic-Rakic Nikunj Mehta Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
{marija,mehta,neno}@usc.edu

ABSTRACT

This paper argues for a set of requirements that an architectural style for self-healing systems should satisfy: adaptability, dynamicity, awareness, autonomy, robustness, distributability, mobility, and traceability. Support for these requirements is discussed along five dimensions we have identified as distinguishing characteristics of architectural styles: external structure, topology rules, behavior, interaction, and data flow. As an illustration, these requirements are used to assess an existing architectural style. While this initial formulation of the requirements appears to have utility, much further work remains to be done in order to apply it in evaluating and comparing architectural styles for self-healing systems.

1 INTRODUCTION

Software architectures provide high-level abstractions for representing the structure, behavior, and key properties of a software system [25]. These abstractions involve descriptions of the elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns [21]. In particular, the system composition patterns and their constraints comprise software *architectural styles*. When designing a software system, selection of appropriate architectural style(s) becomes an important determinant of the system's success. Styles also influence architectural evolution by restricting the possible changes an architect is allowed to make. Examples of styles include pipe and filter, blackboard, client-server [25], GenVoca [3], C2 [26], and REST [8].

Self-healing systems are an emerging class of software systems that exhibit the ability to adapt themselves at run-time to handle situations such as resource variability, changing user needs, and system faults [2]. The topic of self-healing systems has been studied in a number of areas, including robotics and control systems, programming language design, fault-tolerant computing [2], middleware infrastructures [11], and even software architectures (e.g., [6,14]). In particular, [6] provides a technique that explicitly leverages architectural styles to enable system evolution in support of self-healing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WOSS '02, Nov 18-19, 2002, Charleston, SC, USA.

Copyright 2002 ACM 1-58113-609-9/02/0011 ...$5.00

What is currently missing, however, is a general understanding of what constitutes an effective architectural style for self-healing systems in the first place. We believe that such an understanding can help to both streamline existing and develop future architectural techniques in support of this area. This paper presents a first attempt at enumerating requirements that an architectural style for self-healing systems should satisfy: adaptability, dynamicity, awarenessPitM, autonomy, robustness, distributability, mobility, and traceability. Support for these requirements is discussed along five concern areas we have identified in our previous work [17] as distinguishing characteristics of architectural styles: external structure, topology rules, behavior, interaction, and data flow. As an illustration of the application of these requirements on a practical example, we briefly discuss the details of and our preliminary experience with a specific architectural style, called *PitM*, which satisfies a subset of these requirements. Finally, we identify several open issues that will frame our future work.

2 STYLE CHARACTERIZATION FRAMEWORK

An architectural style is a collection of constraints on components, connectors, and their configurations targeted towards a family of systems with shared characteristics. There are many architectural styles currently in use [25]. Styles are often developed systematically to solve difficult architectural problems [9,22,23]; some styles are the result of insights gained from diverse areas of computer science [3,5,12,19,26]. Most styles originate when novel classes of software systems are being designed or existing systems are being adapted to unforeseen circumstances and/or environments. New styles are often produced by changing some aspects of an existing style, or by combining characteristics of multiple existing styles [7].

A systematic understanding of the similarities and differences between different architectural styles is needed to select the most appropriate style for the needs of a given software system. To date, few attempts have been made to provide architectural style comparison and selection tools. Shaw and Clements [24] classify architectural styles into six categories based on their constituent parts, control issues, data issues, and control/data interaction. This study comprises only a preliminary effort to define the underlying characteristics of styles. Some insights into the process of creating a new architectural style (by discussing the differences between a newly codified style and related existing styles) are given in [3,26]. However, these studies fail to provide a systematic basis for developing a novel architectural style that can meet a set of architectural requirements. Additionally, these studies do not provide

enough guidance to designers for creating implementation-level support for architectural styles, thus reducing the utility of styles to conceptual tools for modeling a software system.

Our recent work [17] has focused on analyzing the invariants of various architectural styles such as C2 [26], client/server [5], push-based [9], and pipe-and-filter [25]. In the process, we have found that architectural style invariants can be comprehensively characterized in terms of five concern areas: external structure, topology rules, behavior, interaction, and data flow. *External structure* describes the "shape" of the allowed organizational elements of the style (e.g., components, connectors, ports) and their required and provided services. *Topology rules* specify the allowed composition patterns and paths of interaction between style elements. *Behavior* of style elements defines their internal function, mode of execution, and encapsulated state. *Interaction* between style elements captures the collaboration between architectural elements to collectively achieve the system's purpose. Interaction is specified in terms of four service categories that emerged from our study of software connectors [18]: communication, coordination, conversion, and facilitation. Finally, *data flow* specifies the structure of data exchanged between style elements.

Our hypothesis is that this characterization framework can be used to map architectural requirements to style characteristics, which eventually restrict the design space for the development of a new style. The characterization framework also helps in performing a systematic comparison of various architectural styles that (are claimed to) provide support for a given set of architectural requirements. In Section 3 we demonstrate how this characterization framework can be used to identify the design space of architectural styles for self-healing systems.

3 SELF-HEALING STYLE REQUIREMENTS

Self-healing systems have the ability to modify their own behavior in response to changes in their environment, such as resource variability, changing user needs, mobility, and system faults [2]. Oreizy et. al. [20] describe the lifecycle of self-healing systems as consisting of four major activities:

[1] monitoring the system at run-time,
[2] planning the changes,
[3] deploying the change descriptions, and
[4] enacting the changes.

Based on these activities, the characteristics of existing architectural styles, and the emerging body of work on reflective and self-adaptive systems (e.g., [6,11,14,20]), we have identified the following requirements as pertinent to an architectural style for self-healing systems. Note that additional requirements may be relevant to certain classes of self-healing systems (e.g., support for timeliness). However, in this paper we focus only on those requirements that are likely to be relevant to most such systems. Arguments similar to those provided in the paper may be constructed for other relevant requirements.

- *Adaptability* — The style should enable modification of a system's static (i.e., structural and topological) and dynamic (i.e., behavioral and interaction) aspects. This notion of adaptability does not subsume the issues that must be addressed if such modifications are to be applied during run-time.
- *Dynamicity* — Encapsulates system adaptability concerns during run-time (e.g., communication integrity and internal state consistency).
- *Awareness* — The style should support *reflection* i.e., monitoring of a system's own performance (state, behavior, correctness, reliability, and so forth) and recognition of anomalies in that performance. The style should also support *observability* i.e., monitoring of the system's execution environment. Note that the system may not be able to influence changes in its environment (e.g., reestablishing failed network links), but it may plan changes within itself in response to the environment (e.g., performing in a degraded mode until the network link is reestablished).
- *Autonomy* — The style should provide the ability to address the anomalies (discovered through awareness) in the performance of a resulting system and/or its execution environment. Autonomy is achieved by planning, deploying, and enacting the necessary changes.
- *Robustness* — The style should provide the ability for a resulting system to effectively respond to unforeseen operating conditions. Such conditions may be imposed by the system's external environment (e.g., malicious attacks, unpredictable behavior of the system's run-time substrate, unintended system usage), as well as errors, faults, and failures within the system itself. Note that this definition of robustness subsumes fault-tolerance.
- *Distributability* — The style should support effective performance of a resulting system in the face of different distribution/deployment profiles.
- *Mobility* — The style should provide the ability to dynamically change the (physical or logical) locations of a system's constituent elements.
- Traceability — The style should clearly relate a system's architectural elements to the system's execution-level modules in order to enable change enactment in support of the above requirements.

Table 1 shows a mapping of the listed requirements to the five style characteristics identified in Section 2.[1] Each requirement is related to one or more of the four major activities in a self-healing system's lifecycle discussed above. This characterization represents our initial attempt at classifying the space of architectural styles for self-healing systems. While we have identified this decomposition based on the properties of a large number of existing architectural styles, we have tried to characterize the requirements independently of any specific architectural style. Note that our goal is not to present in the table all possible ways in which a style may satisfy a particular requirement. Rather, we attempt to provide a set of representative solutions.

Some requirements (e.g., *adaptability*, *dynamicity*) impose a set of basic structural and data-flow elements. These elements are subsequently leveraged to support other requirements. In support of *dynamicity* and *mobility* we have identified the need for specialized structural elements (referred to as *effectors* in Table 1), which are capable of enacting the desired changes in the architecture. We also propose the use of *analysis agents* whose role is to ensure the validity of the desired changes (e.g., preservation of the style's topological rules).

1. Traceability is an architectural process that influences implementation-level support. Therefore, this requirement is not captured in the characterization provided in Table 1.

Requirement (activities) \ Characteristics of the style	Structure	Topology	Behavior	Interaction	Data flow
Adaptability (enacting changes)	Separable components ✓ Explicit connectors ✓ Explicit entry/exit portals ✓ Primitive ducts ✓	Portals attached to a single component or connector ✓ Constrained number of component portals - limited component dependencies ✓ Expandable (number of) connector portals ✓ Connectivity pattern exclusively portal[a]-duct-portal[b], where both portals cannot belong to components ✓ Limited component dependencies (visibility) ✓	Exposed via named services only ✓	Asynchronous coordination ✓ Implicit invocation ✓ Event-based interaction ✓ Adjustable connector delivery policies ✓	Discrete events ✓ Data streams
Dynamicity (enacting changes)	Separable components ✓ Explicit connectors ✓ Explicit entry/exit portals ✓ Primitive ducts ✓ Dynamism effectors ✓ Dynamic change analysis agents ✓	Components and connectors dynamically created ✓ Modifiable portal-to-portal bindings (i.e. ducts) ✓ Dynamically expandable (number of) connector portals ✓ Less constrained topological rules for attaching dynamism effectors to application architecture (to add the possibility of more direct control over the architecture) ✓ Dynamism effectors attached to analysis agents to ensure desired system properties during the change ✓	State preservation/restoration Component quiescence Dynamism change analysis ✓ Data queueing and buffering by connectors ✓ Dynamism effecting functionality ✓	Different interaction categories (e.g., allowed, deferred, disallowed) ✓ Delivery guarantees (at least once, exactly once) ✓ Synchronous, possibly real-time, meta-level dynamic change requests ✓	Dynamic change events ✓ System architectural models used to analyze the validity of proposed changes ✓ State transfer events
Awareness (monitoring)	Introspection portals Introspection interfaces ✓ Meta-level components ✓ Meta-level connectors Self-monitors System monitors ✓	Less constrained topological rules for attaching to introspection facilities, to enable a more direct control over the architecture ✓ Direct binding of monitors to components, connectors, portals, or ducts ✓ Less constrained topological rules for attaching to environment facilities	Self-monitoring and assessment functionality ✓ Execution trace capture ✓ Execution trace analysis Environment-monitoring and assessment functionality Environment event analysis	Real-time system monitoring data delivery (to monitor correctness) ✓ Asynchronous system monitoring data delivery (to monitor statistical performance) Ongoing or intermittent environment monitoring	System monitoring events ✓ Critical system monitoring event patterns Environment-level events Critical environment-level event patterns
Autonomy (planning, deploying, enacting changes)	Autonomous meta-level components ✓ Explicit, autonomous connectors ✓	Meta level components connected to dynamism effectors and change analysis components ✓	Adjustable planning policies	Synchronous architectural dynamism (to ensure that the change is atomic)	Dynamic change events ✓ System architectural models ✓ State transfer events
Robustness (planning, deploying, enacting changes)	Autonomous components ✓ Explicit entry/exit portals ✓ Primitive ducts ✓	Connectivity only via (known) portals and ducts ✓	Exception handling Data queueing/buffering ✓	Asynchronous coordination ✓ Implicit invocation ✓ Event-based interaction ✓	Discrete events ✓ Exception propagation
Distributability (general requirement)	Autonomous components ✓ Explicit and distributed connectors ✓ Explicit portals to remote environments ✓ Distribution channels (ducts) ✓ Distributed node registries	Distributed topology rules same as local topology rules ✓	Data caching by distributed connectors ✓ Connection setup and teardown Multi-tasking mechanisms such as threads ✓	Remote procedure calls (RPC) Discrete event-based interaction ✓ Continuous stream-based interaction Data marshalling and unmarshalling by distributed connectors ✓ Adjustable scheduling policies ✓ Delivery guarantees ✓	Byte streams Discrete events ✓ Quality of interaction parameters (e.g., real-time constraints, delivery guarantees, security, synchronicity) ✓
Mobility (deploying, enacting changes)	Separable components ✓ Distributed connectors ✓ Explicit portals to remote environments ✓ Modifiable portal-to-duct bindings ✓ Mobility effectors ✓ Mobility effect analysis agents ✓	Modifiable portal-to-duct bindings ✓ Limited component dependencies (visibility) ✓ Dynamically expandable (number of) connector portals ✓ Less constrained topological rules for attaching mobility effectors to application architecture ✓ Mobility effectors attached to analysis agents ✓	State transfer Component quiescence Data queueing and buffering by connectors ✓ Mobility effecting functionality ✓	Different interaction categories during migration (e.g., allowed, deferred, disallowed) ✓ Delivery guarantees (at least once, exactly once) ✓ Data rerouting to new destinations	Meta-level mobility requests ✓ Data tuples System components ✓ System architectural models ✓

Table 1: Requirements for self-healing systems and their mapping to the five style characteristics. ✓denotes support in PitM.

The characterization provided in Table 1 can be used to create a new architectural style for self-healing systems. Selections of the requirements to be supported and of specific values for the facets of those requirements will identify a specific architectural style for self-healing systems. This characterization may also be used to evaluate and compare a set of architectural styles, and to select the most appropriate style for a given system in the self-healing domain. We should note that such an evaluation and/or comparison would be premature at this time as our understanding of the domain of self-healing systems is still evolving. For example, our intuition is that it may not be necessary for a style to be able to support *all* of the requirements listed in Table 1; however, we need much additional experience in order to verify that intuition.

4 ASSESSMENT OF THE PITM STYLE

Over the past three years, we have worked on the design of a specific architectural style, called *PitM*, for supporting self-healing systems in highly distributed, mobile, and resource-constrained environments [15]. In formulating the PitM style, we have leveraged our extensive experience with the C2 architectural style, which is intended to support highly distributed applications in the GUI domain [26]. In this section, we provide an assessment of PitM along the dimensions established in Section 3. Note that the entries in Table 1 that are supported by the PitM style are marked.

The PitM style supports the following basic **structural elements**.

[1] *Components*, which maintain state and perform application-specific computation;

[2] Explicit *connectors*, which mediate all component interactions. PitM distinguishes between two types of connectors: a *horizontal* connector enables the asymmetric communication among components through their top and bottom ports, while a *peer* connector enables symmetric communication among components through their side ports (see Figure 1);

[3] Explicit communication *ports*: each component has three ports (named top, bottom, and side), while connectors contain an adjustable number of ports for attaching components; and

[4] Primitive *ducts* [18] for connecting ports.

Basic **data flow** in the PitM style is achieved via *events* (also referred to as *messages*). A message in the PitM style is either a *request* for a component to perform an operation, a *notification* that a given component has performed an operation and/or changed its state, or a *peer* message used in symmetric (peer-to-peer) communication between components. Request messages are sent through the top port, notifications through the bottom port, and peer messages through the side port of a component (see Figure 1).

Basic **topology rules** of the PitM style mandate that:

[1] each component port may be attached to at most one connector port,

[2] the number of connector ports is adjustable and unlimited in principle, and

[3] a component in an architecture may have no knowledge of or dependencies on components below it.

Furthermore, two PitM components may not engage in interaction via peer messages if there exists a vertical topological relationship between them (through direct or indirect connectivity via horizontal connectors). For example, components *B* and *D* in Figure 1 may not exchange peer messages since one component is above the other; on the other hand, no vertical topological relationship can be established between components *C* and *D*, so that they may communicate via peer messages. For a similar reason, the PitM style disallows the possibility of exchanging messages between a peer and a horizontal connector (which would, in effect, convert peer messages into requests/notifications, and vice versa).

Component **behavior** in the PitM style is exposed via a set of provided and required services, while the **interaction** is achieved using both asynchronous and synchronous event-based communication. Invocation of components is implicit (via named events).

These basic characteristics of the PitM style are intended to address the **ADAPTABILITY** requirement, and to enable support for other requirements discussed in Section 3. In the remainder of this section we highlight several additional characteristics of the PitM style that are targeted at supporting the requirements identified in Section 3.

To support **DISTRIBUTABILITY**, the PitM style natively provides connectors that span device boundaries. Such connectors, called *border connectors*, enable the interactions of components residing on one device with components on other devices. A border connector marshals and unmarshals data, code, and architectural models, and dispatches and receives messages across the network. It may perform data compression for efficiency, and authentication, authorization, and encryption for security. Finally, a border connector supports different message delivery guarantees (e.g., exactly once, best effort).

In order to address the need for adapting highly dynamic and mobile applications to changes in their execution context, the PitM style supports architectures at two levels: application-level and meta-level. The role of components at the PitM meta-level is to act as *effectors* and facilitate **DYNAMICITY**, **AWARENESS**, **MOBILITY**, and **ROBUSTNESS** of PitM applications. Meta-level components may be application-independent or application-specific. Application-level and meta-level components exist side-by-side in PitM. Meta-level components are aware of application-level components and may initiate interactions with them, but not vice versa. The PitM style rules apply to both component categories: meta-level components

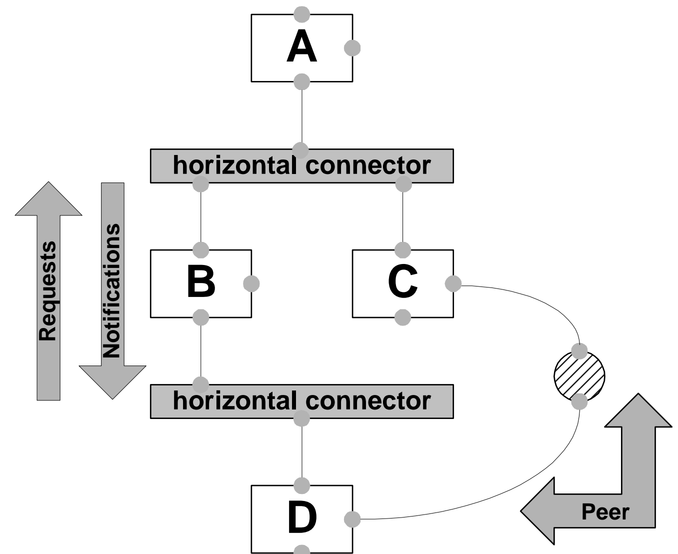

Figure 1. A simple application architecture in the PitM style. The unlabeled shaded circle represents a peer connector.

also engage in connector-mediated, message-based interactions with each other (and with application-level components).

In support of this two-level architecture, PitM currently distinguishes among four types of messages. *ApplicationData* messages are used by application-level components to communicate during execution. The other three message types, *ComponentContent*, *ArchitecturalModel*, and *SystemMonitoring* are used by PitM meta-level components (i.e., effectors). *ComponentContent* messages contain mobile code and accompanying information (e.g., the location of a migrant component in the destination configuration); *Architectural-Model* messages carry information needed to perform architecture-level analyses of prospective PitM configurations; finally *SystemMonitoring* messages carry the information about different aspects of the system's performance.

We have extensively employed special-purpose meta-level components, called *Admin Component*s, whose task is to exchange *ComponentContent* messages and facilitate the deployment and migration of application components across devices. Another meta-level component is the *Continuous Analysis* component, which leverages *ArchitecturalModel* messages for analyzing the (partial) architectural models during the application's execution, assessing the validity of proposed run-time architectural changes, and possibly disallowing the changes.

Finally, **TRACEABILITY** is ensured in PitM via an architectural middleware implementation [16] that maintains a strict relationship between PitM architectural elements and implementation-level modules. This middleware leverages programming language-level techniques such as introspection and dynamic loading in order to support the corresponding stylistic requirements.

We have recently begun adding to PitM support for observability (i.e., awareness of the system's execution environment) via execution environment monitors and extensions to border connectors. Currently, PitM's support for **AUTONOMY** is limited to acting upon anticipated (i.e., pre-coded) operational anomalies.

5 SUMMARY AND OPEN ISSUES

In this paper we have presented our attempt at codifying the requirements for an architectural style in the domain of self-healing systems. These requirements are needed to effectively support the four major activities in the self-adaptive software lifecycle: monitoring, planning the changes, deploying change descriptions, and enacting the changes [20]. The requirements have been discussed along the five concern areas we have identified previously as distinguishing characteristics of architectural styles [17]. Such characterization provides a clear mapping of the identified requirements to the style dimensions. Additionally, it provides comparison criteria for architectural styles in the self-healing domain. Our hope is that the selection of requirements to be supported and values for their characteristics will help to identify specific self-healing architectural styles in a straightforward way.

This work is preliminary and much remains to be done. A natural next step is to gain further experience by evaluating the applicability of a number of known architectural styles to the self-healing domain using the framework outlined in this paper. Such an evaluation will, in turn, be used to fine-tune the framework itself. In addition, we are currently investigating the suitability of various formalization techniques, some of which have already been applied at the level of architectural styles [1,4,10,13], for streamlining and possibly automating the task of self-healing style evaluation and comparison. The application of our framework to the PitM style in Section 4 suggests the need for formalization: the various (specific) aspects of PitM were formulated in a similar, but not identical, way to the (more general) descriptions given in Table 1; in order to establish whether and to what extent those aspects of PitM truly are similar to the table entries, ideally one would be able to formally specify and check for a relationship akin to subtyping between the two. While formal specification and analysis of architectural styles is by no means a trivial task, our recent experience [17] suggests that this may indeed be accomplished.

Our future work on the PitM style itself will focus on more general support for observability (which is critical in resource-constrained, mobile environments) and autonomy (which is central to the concept of self-healing). We believe that work by Cheng et al. [6] suggests some interesting possibilities in this regard, which we intend to explore. We also plan to extend PitM's current dynamicity and mobility support with state transfer, and state preservation/restoration facilities. Finally, we intend to map all of these capabilities to PitM's implementation infrastructure.

6 ACKNOWLEDGEMENTS

This material is partly based upon work supported by the National Science Foundation under Grant No. CCR-9985441. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement number F30602-00-2-0615. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government. Effort also sponsored in part by the U.S. Army Tank Automotive and Armaments Command and Xerox Corporation.

7 REFERENCES

[1] G. D. Abowd, R. Allen and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4): 319-364, October 1995.

[2] ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02) Call for Papers. http://www-2.cs.cmu.edu/~garlan/woss02/

[3] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology,* 1(4):355-398, October 1992.

[4] M. Bernardo, P. Ciancarni and L. Donatiello. On the formalization of architectural types with process algebras. *Proceedings of the 8th International Symposium on Foundations of Software Engineering*, pp. 140-148, November 2000.

[5] A. Berson. Client/Server Architecture. *McGraw-Hill*, 1992.

[6] S. Cheng et. al. Using Architectural Style as a Basis for Self-repair. *Proceedings of the 2002 Working IEEE/IFIP Conference on Software Architectures (WICSA 2002)*, Montreal, Canada, August 25-30, 2002.

[7] N. Delisle and D. Garlan. Applying Formal Specification to Industrial Problems: A Specification of an Oscilloscope. *IEEE Software*, 7(5):29-37, September 1990.

[8] R. Fielding. Architectural Styles and the Design of Network-Based Software Architectures. *Ph.D Thesis*, UC Irvine, June 2000.

[9] M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. In *Proceedings of the 7th European Software Engineering Conference*, pp. 20-38, October 1999.

[10] D. Jackson and K. Sullivan. COM revisited: Tool assisted modelling and analysis of software structures. *Proceedings of 8th ACM SIGSOFT Symposium on Foundations of Software Engineering*, San Diego, CA, November 2000.

[11] F. Kon, et. al.The Case for Reflective Middleware. *Communications of the ACM*. 45(6):33 - 38, June 2002.

[12] G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26-49, August/September 1988.

[13] D. Le Metayer. Describing architectural styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7): 521-533, July 1998.

[14] A. Lopes et. al. Architectural Primitives for Distribution and Mobility. *Proceedings of Symposium on Foundations of Software Engineering - 10*, Charleston, South Carolina, November 2002.

[15] N. Medvidovic and M. Mikic-Rakic. Architectural Support for Programming-in-the-Many. *TR USC-CSE-2001-506*, USC, 2001.

[16] N. Medvidovic, N. Mehta, and M. Mikic-Rakic. A Family of Software Architecture Implementation Frameworks. *Proceedings of the 2002 Working IEEE/IFIP Conference on Software Architectures (WICSA 2002)*, Montreal, Canada, August 25-30, 2002.

[17] N. Mehta. Distilling Software Architectural Primitives from Architectural Styles. *TR USC-CSE-2002-504*, USC, 2002.

[18] N. Mehta, N. Medvidovic and S. Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, May 2000.

[19] H. P. Nii. Blackboard Systems. *AI Magazine,* 7(3):38-53 and 7(4):82-107, 1986.

[20] P. Oreizy et. al. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999. pages 54-62.

[21] D.E. Perry, and A.L. Wolf. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.

[22] N. Roodyn and W. Emmerich. An Architectural Style for Multiple Real-Time Data Feeds. In *Proceedings of the 21st International Conference on Software Engineering,* pp. 564-572, Los Angeles, CA, US, June 1999.

[23] D. S. Rosenblum and A. L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proceedings of the 6th European Software Engineering Conference,* Zurich, Switzerland, September 1997.

[24] M. Shaw and P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of 21st Annual Computer Software and Applications Conference*, pp. 6-13, 1997.

[25] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice-Hall*, 1996.

[26] R. N. Taylor, et. al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390-406, 1996.