Software Architectures and Embedded Systems

Nenad Medvidovic Sam Malek Marija Mikic-Rakic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
{neno,malek,marija}@usc.edu

Introduction

Software architecture has emerged as an area of intense research over the past decade [25,32]. A number of approaches have been proposed to deal with architectural description and analysis [21], architectural styles [8], domain-specific and application family architectures [4,35], architecture-based dynamic system adaptation [29], and so forth. By and large, however, these approaches share assumptions that make them suited specifically to the domain of traditional, desktop-based, possibly distributed development platforms. Those (comparatively few) architecture-based solutions that have focused on software systems for embedded devices (e.g., [28]) have had to face some of the same challenges (e.g., applying solutions across an application family), but also appear to have had some different priorities (e.g., ensuring efficient, architecture-compliant system implementations).

The goal of this paper is to draw some general distinctions between "traditional" software architectures and those targeted at embedded systems. We focus on several areas that traditional software architecture research has studied to date and discuss their applicability and potential shortcomings in the context of embedded systems. As suggested above, our position is informed by the existing literature. Additionally, we will leverage insights drawn from a graduate course we have offered at USC for the past two years—*Software Engineering for Embedded Systems* [35]. Finally, we will also rely on the experience from *Prism*, an on-going research project whose goal is to develop software architectural solutions for the domain of highly distributed, mobile, resource-constrained, and possibly embedded computation [19,23]. The discussion provided in this paper should not be considered a definitive study of this issue, but rather a starting point for future discussions. We also attempt to provide a critical assessment of our Prism project with respect to the discussed areas, in the hope of outlining future directions that will improve Prism's suitability for the embedded systems domain.

Architectural Modeling

A large number of special-purpose architecture description languages (ADLs) [21] have been developed to represent different aspects of software architectures. More recently, the Unified Modeling Language (UML) [3] has gained widespread acceptance for a similar purpose. Table 1 shows an overview of several ADLs and their primary foci. Only two of these notations are specifically intended for the embedded systems domain:

1. MetaH models architectures in the guidance, navigation, and control (GN&C) domain. MetaH tries to ensure the schedulability, reliability, and safety of the modeled software system. It also considers the availability and properties of hardware resources.
2. Weaves [10] supports specification of data-flow architectures. In particular, Weaves is specialized to support real-time processing of large volumes of data emitted by weather satellites.

Table 1: – ADL Scope and Applicability

ADL	ACME	Aesop	C2	Darwin	MetaH	Rapide
Focus	Architectural interchange at the structural level	Specification of architectures in specific styles	Architectures of distributed, evolvable systems in a particular style	Architectures of distributed, dynamic systems	Guidance, navigation, and control system architectures	Modeling and simulation of the dynamic behavior of an architecture

ADL	ROOM	SADL	UniCon	Weaves	Wright
Focus	Graphical (structural and behavioral) models of architectures	Formal refinement of architectures across levels of detail	Glue code generation for interconnecting existing components	Data-flow architectures with real-time requirements on data processing	Modeling and deadlock analysis of concurrent systems

Two other notations also deal with issues that are relevant to the embedded systems domain: UniCon [32] supports modeling of runtime (though not necessarily real-time) scheduling, while ROOM [31] targets real-time computation with a combination of message sequence charts and state charts. The more recent Avionics ADL [5] tries to marry several of these ideas into a single language.

Despite the above examples, many questions remain unanswered. It is unclear whether modeling the architectures of embedded systems is inherently incompatible with semi-formal notations such as UML. The prevailing characteristics of the embedded systems domain would suggest that rigor and formality are a non-negotiable requirement. On the other hand, counter-examples exist. For instance, the software architecture team working on JPL's Mission Data Systems (MDS) architectural framework [39] had initially selected UML for representing MDS architectures; the team has recently replaced UML with an XML-based ADL [8], but one that still has only semi-formally defined semantics.

Another relevant issue is deciding which aspects of an embedded software system are critical from an architectural perspective. It is widely agreed that components, connectors, and their configurations are the basic building blocks of a traditional software architecture. While the same claim might be made about embedded system architectures, embedded systems have certain characteristics that require careful consideration. For example, a to-be-embedded software system is often built to a specification, with the actual platform(s) being unknown or even non-existent at the time of development. In such cases, an application domain-specific ADL (e.g., MetaH for the GN&C domain) may help in identifying at least those system aspects that are likely to remain stable across specific applications and platforms.

We have not focused on architectural modeling in the Prism project. Instead, we have chosen to rely on our existing architecture modeling infrastructure [20] in order to address other issues, as discussed below.

Analysis

A particular focus of existing ADLs has been on formal analysis of system properties at the architectural level. However, ADLs are hampered by two problems, both of which are likely to be further magnified in the case of embedded systems. The first issue is analysis *fidelity*: while languages such as Wright [1] allow modeling and sophisticated analyses of dynamic system prop-

erties (e.g., potential for deadlocks), they either make large numbers of simplifying assumptions about, or fail to consider altogether, the capacity, speed, power, and other properties of the hardware platforms on which the modeled software systems will execute. In addition, most existing ADLs provide scant support for transferring the desired architectural decisions into source code. This is absolutely critical in the case of embedded systems, where an elegant and "correct" software architecture will be of little use unless it results in an effective and efficient *running* system.

The lack of analysis fidelity suggests the second issue that has not been adequately addressed in existing ADLs. In order to gain confidence that the system will behave correctly in its target environment, *simulation* of that system model's execution behavior in the (simulated) environment becomes indispensable. While there is a lot of potentially relevant work on simulation, this work has taken place almost entirely outside the domain of software architectures. The lone exception to this is Darwin [16] which leverages its π-calculus underpinnings to support execution of "what if" scenarios. Much additional work is needed if ADLs are to be rendered suitable for use with embedded systems in this regard.

As in the case of modeling, thus far in the Prism project we have not particularly focused on architectural analysis of embedded system architectures.

Architectural Styles and Reference Architectures

Software *architectural styles* are recurring patterns of system organization whose application results in systems with known (desirable) properties [9,33]. As such, styles are key software design idioms. Examples of well known styles are layered, pipe-and-filter, client-server, push-based, peer-to-peer, event-based, and so forth. At the same time, very little is known about the applicability of these, or any other styles, to the embedded systems domain. There are a few exceptions to this (e.g., [12,38]), but they have emerged from problem domains (e.g., mobile robotics) in which software engineering issues, and software architectures in particular, are considered to be of secondary importance.

An issue related to styles is that of *reference architectures*. A reference architecture is applicable to systems across an application family and/or problem domain. Unlike a style, which provides a set of heuristics for arriving at a software system's architecture, a reference architecture only needs to be instantiated into a system architecture (i.e., it is already an architecture, but a generic one). Even though effective styles for embedded systems may be unknown, there are examples of successful reference architectures in this area. One such example is Phillips's Koala project [28], which is targeted at consumer electronics devices. Another example is IBM's ADAGE [6] reference architecture (illustrated in Figure 1 below), targeted at the avionics domain. Reference architectures are hard to come by because they require (extensive) existing experience within an application domain, but, once in existence, they do appear to be a natural fit with embedded systems.

We have placed special emphasis on architectural styles in the Prism project. Since, as discussed above, there is currently very little understanding of styles that are appropriate in the embedded systems setting, we have developed an architectural style *framework*, called *Prism-SF* [19], which may be instantiated into a number of individual styles. The framework fixes a small number of architectural notions: computation is performed within (autonomous) *components*, and their interaction enabled via *events*. All other architectural concepts (e.g., software *connectors*, communication *ports*) are available and configurable, but not required. Finally, Prism-SF allows the

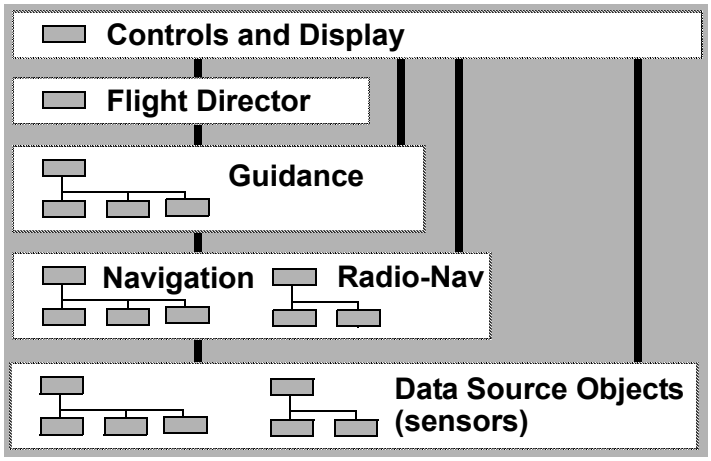

Figure 1. ADAGE – a five-layer reference architecture intended for systems in the avionics domain. Every ADAGE-compliant system will have the depicted five layers. Each of the five layers may, in turn, comprise its own layered internal architecture.

selection and combination of four types of interaction: symmetric (i.e., peer-to-peer) and asymmetric (i.e., master-slave), as well as synchronous and asynchronous. These facilities provided by Prism-SF are instantiated into one or more architectural styles, which are, in turn, used in the design of specific systems. For illustration, Figure 2 shows a partial application architecture designed using an instance of Prism-SF in which

- both symmetric and asymmetric connectors are included;
- each component has single master, slave, and peer communication ports; and
- symmetric and asymmetric connectors may not be attached to one another.

Implementation Support

Software architectures provide design-level models and guidelines for composing software systems. For these models and guidelines to be truly useful in a development setting, they must be accompanied by support for their implementation [18,32]. This is particularly important in the

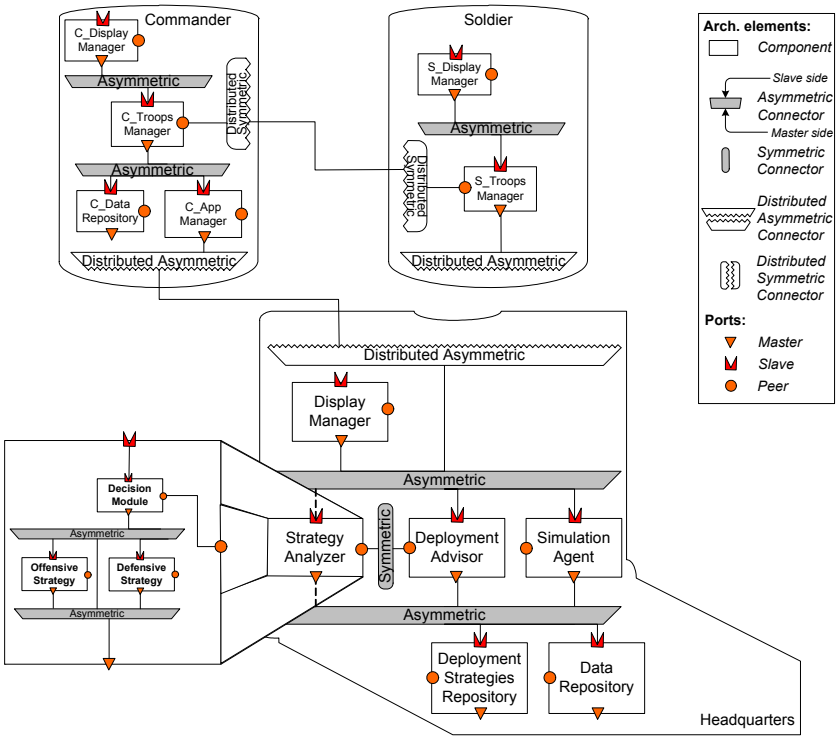

Figure 2. An application architecture designed using a particular instantiation of the Prism-SF architectural style framework.

context of embedded systems: they may be highly distributed, decentralized, mobile, and long-lived large-scale software systems, increasing the risk of architectural drift [25] unless there is a clear relationship between the architecture and its implementation. Recent studies [23,32] have shown that an effective way to realizing the system's architecture is to leverage the support provided by architectural middleware solutions. Typically an architectural middleware provides implementation-level support for the key aspects of system's architecture: components, connectors, architectural configurations, and communication events.

Embedded systems are usually characterized by resource constraints. However, middleware solutions introduce an abstraction layer and therefore raise the issue of its effect on the system's performance. For a middleware platform to be usable in the context of embedded systems, it needs to be highly efficient. Depending on the nature of the embedded environment, efficiency can entail minimum use of CPU, memory, battery power, network bandwidth, and so on. Recently, some vendors of popular middleware solutions (e.g., CORBA Orbix [13]) have started to tailor their support for use in the embedded systems domain. Table 2 below compares middleware solutions for mobile, resource constrained, possibly embedded systems along several pertinent dimensions. The table has been adapted from [23].

Table 2: Comparison of existing middleware solutions. **?** denotes unavailable data; ✓✓✓ denotes extensive support; ✓✓ denotes solid support; ✓ denotes some support; empty cells denote no support.

Property	*Orbix/E* [13]	*TAO* [30]	*JXTA* [26]	*.NET* [22]	*JINI* [34]	*XMIDDLE* [17]	*RCSM* [40]	*LIME* [15]	*Prism-MW* [23]
Architectural abstractions									✓✓✓
Efficiency [a]	16.6K	8K	?	?	?	?	?	?	20K
	95KB	0.5MB	?	?	?	156KB	?	?	4.6KB
Scalability	✓✓	✓✓	✓✓	✓✓	✓✓	?	?	?	✓✓✓
Delivery guarantees	✓	✓✓✓							✓
Mobility					✓✓	✓✓✓	✓✓	✓✓✓	✓✓
Reconfigurability		✓✓			✓✓		✓✓	✓✓	✓✓
Security	✓✓	✓✓	✓✓	✓✓	✓		✓✓		✓

a. Number of events per second (top) and memory usage (bottom).

Heterogeneity is another intrinsic characteristic of embedded systems [14,19]: unlike traditional software platforms, which have been standardized to a significant degree, many embedded systems run on one-of-a-kind hardware with special-purpose operating systems, programming languages, network protocols, data formats, and so forth. This poses a great challenge to embedded application developers. Techniques commonly employed to address heterogeneity in traditional software systems, such as XML encoding or platform independent programming languages (e.g., Java), also may not be viable options in the embedded systems domain due to resource scarcity and possible real-time requirements. Therefore, a practical architectural middleware solution in this domain needs to be either highly specialized (and thus narrowly applicable) or flexible in order to overcome the unpredictable and the heterogeneous nature of embedded systems.

Our implementation support for software architectures in the Prism project is embodied in the *Prism-MW* middleware platform [23]. Prism-MW provides implementation-level support for key abstractions provided by the Prism-SF architectural style framework discussed above. Prism-

Architecture initialization

```
class DemoArch {
  static public void main(String argv[]) {
  Architecture arch = new Architecture ("DEMO");
    // create components here
    ComponentA a = new ComponentA ("A");
    ComponentB b = new ComponentB ("B");

    // create connectors here
    Connector conn = new Connector("Conn");

    // add components and connectors to the architecture
    arch.addComponent(a);
    arch.addComponent(b);
    arch.addConnector(conn);

    // establish the interconnections
    arch.weld(a, conn);
    arch.weld(b, conn);
    arch.start();
  }
}
```

Component A sends an event

```
  Event e = new Event ("Event_a");
  e.addParameter("param_1", p1);
  send (e);
```

Component B handles the event and sends a response

```
public void handle(Event e)
{
  if (e.equals("Event_a")) {
    ...
    Event e1= new Event("Response_to_a");
    e1.addParameter("response", resp);
    send(e1);
  }...
}
```

Figure 3. Illustration of application implementation fragments in Prism-MW. The created simple architecture has two components, *A* and *B*, communicating via a connector, *Conn*.

MW is, at the same time, optimized to exhibit a small memory footprint and good system speed (see Table 2), and extensible to address many relevant concerns, including distribution, mobility, security, data compression, and so forth. However, Prism-MW currently does not consider other hardware resources, such as battery power, or availability and properties of peripheral devices. Figure 3 illustrates how an architecture is "programmed" in the Java version of Prism-MW.

Deployment Support

An embedded software system is typically developed and tested in a simulated environment. The target hardware environment is frequently produced in parallel with the software system, and therefore may not be available before or during the software system's development and testing. Alternatively, the actual target environment may be too expensive to replicate or it may be too distant (e.g., a space probe). However, the characteristics of the target environment directly influence certain software decisions, such as the distribution of software components onto hardware hosts (i.e., the system's deployment architecture). Furthermore, the target environment often changes during the system's execution (e.g., due to the mobility of hardware hosts). As a result of this, there is an increasing demand for deployment support that can assist with the installation and/or update of a software system. Traditional approaches to software deployment have often required sophisticated support (e.g., deployment agents [11]). These approaches have typically included their own facilities to inspect the target environment prior to deployment. Since these facilities are provided separately from the application's implementation infrastructure, they introduce addi-

tional overhead to the target host. Therefore, these approaches are usually not directly applicable for resource constrained, embedded software systems.

Mainstream software deployment solutions have comprised large-scale "patches" that replace an entire application or set of applications (e.g., new versions of MS Office). This kind of coarse-grain deployment does not provide sufficient control over the deployment process and is usually not applicable to distributed embedded systems with low-bandwidth network connections through which the patches need to be exchanged. For these reasons, efficient, fine-grain control over the deployment process is required in the context of embedded systems.

Finally, in highly distributed embedded systems, deployment and/or update process may need to be initiated from multiple sites, each of which is in charge (and possibly aware) of only a part of the overall system. Furthermore, the source locations of software components that need to be installed or updated may be themselves distributed, requiring efficient support for exchanging the necessary software components between the source and destination hosts.

Current support for deployment in the Prism project is accomplished via an extension to MS Visio and a "skeleton" configuration in Prism-MW consisting of a special purpose *Admin* component and a distribution-enabled connector. The resulting tool, *Prism-DE*, is shown in Figure 4. Prism-DE allows one to specify a configuration of hardware hosts (with known IP addresses), operating system processes on each host, and software configurations (comprising software components and connectors) within each process. Once a suitable deployment is depicted, it is effected with a single button push. The *Admin* components on each host receive and instantiate (using Prism-MW's API shown in Figure 3) the application-specific components. Further details of this process are discussed in [24]. Prism-DE then continuously monitors the network connectivity of the depicted hardware configuration. As indicated in the above discussion, Prism-DE currently sup-

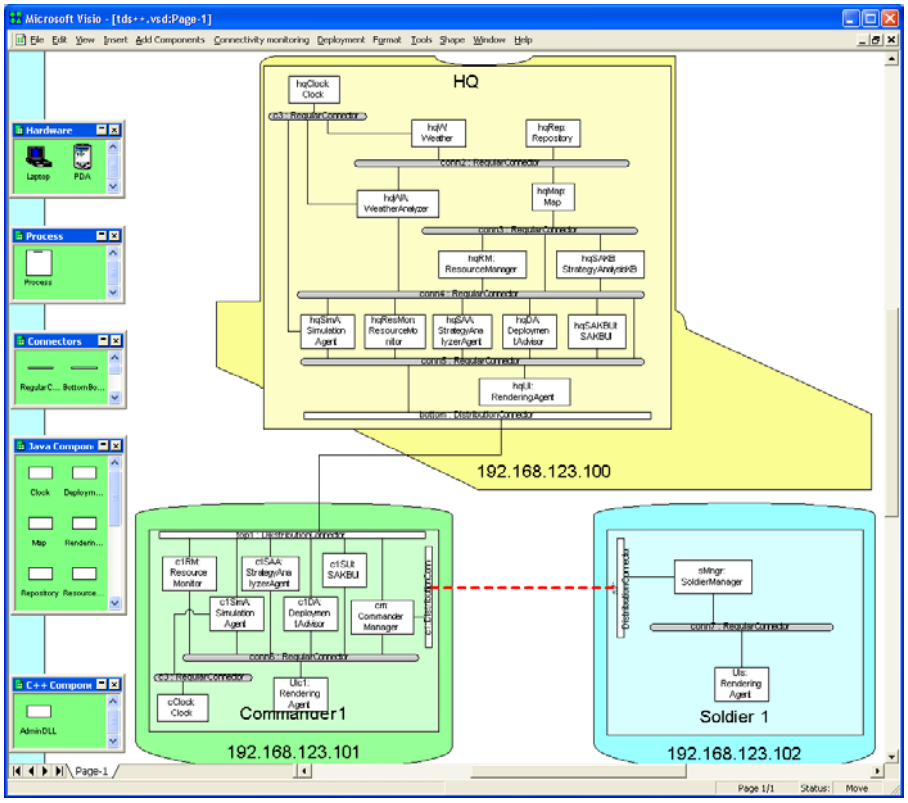

Figure 4. Screenshot of the Prism-DE deployment environment. The dotted line between the two bottom device icons denotes a network disconnection.

ports only *centralized* deployment, in which the entire system's deployment architecture, as well as the locations of all component implementations are known *a priori*.

Dynamic Adaptability

Many embedded systems are safety-critical systems that concurrently engage the physical world. These systems must be capable of adapting to changes in their execution environment. However, usually these systems cannot be brought down for updates [32]. Dynamic adaptability provides a solution to this problem, as it enables one to modify a running software system without stopping its execution. However, system properties (e.g., availability, safety) may get affected during the system's dynamic manipulation and therefore need to be accounted for when performing the adaptation. This can be achieved by analyzing (both statically [1] and dynamically [27]) the likely effects of the proposed changes before they are enacted.

Dynamic adaptation may modify the system's architecture. These modifications need to be captured and maintained at the architectural level to ensure consistency between the architectural model and the running system. Depending on the origin of changes and the degree of distribution of the software system, the task of maintaining the consistency may be trivial (i.e., updating the architectural model before initiating the change or after the change is completed) or highly complex (if there are many changes on many target hosts, the information about these changes needs to be propagated to the host maintaining the architectural model). Furthermore, in decentralized embedded systems there may not be a single host capable of maintaining the system's overall architectural model. We are aware of no existing solutions for ensuring correct, consistent, and safe dynamic adaptability of such systems.

As indicated in the above discussion, the current support for dynamic system adaptability in Prism assumes a centralized architectural model of the system. We leverage Prism-MW's API for adding, removing, and reconnecting components in an architecture (recall Figure 3), as well as programming language-level facilities (e.g., dynamic class loading in Java) and operating system-level facilities (e.g., DLLs in Windows), to enable dynamic system manipulation at the software component level. While system analysis prior to dynamic change has not been a particular focus of our work to date, as a "proof of concept" we have provided special purpose *Architectural Analysis* components within Prism-MW. These components implement a variation of our static analysis capabilities for software architectures [20].

Conclusion

It has been claimed that software architectures have the potential to revolutionize large-scale software development by allowing developers to shift their focus away from a system's implementation details. However, in the case of embedded systems, the devil is indeed in the details: a number of implementation-level issues have direct implications on a system's success and even its viability. These issues, therefore, must be captured and properly assessed at the level of an embedded system's software architecture; otherwise the architecture will be of little use. In this paper, we have used the traditional "playing field" of software architectures to highlight some of the unique challenges introduced by the embedded systems domain and briefly introduce emerging techniques to address these challenges; we have also provided an evaluation of our Prism project with respect to these challenges. We hope to have provided a useful first step in what is surely a topic worthy of further study.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213-249, July 1997.

[2] D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE Avionics Reference Architecture. *A1AA Computing in Aerospace-10*, San Antonio, Texas, March 28-30 1995.

[3] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide. *Addison Wesley*, 1999.

[4] J. Bosch. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. *Addison-Wesley (Pearson Education)*, May 2000.

[5] E. Colbert, B. Lewis, and S. Vestal. Developing Evolvable, Embedded, Time-Critical Systems with MetaH. *34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 34)*, Santa Barbara, August 2000.

[6] L. Coglianese and R. Szymanski, DSSA-ADAGE: An Environment for Architecture-based Avionics Development, *In Proceedings of AGARD,* 1993.

[7] D. Daniel, R. Rasmussen, G. Reeves, A. Sacks. Software Architecture Themes In JPL's Mission Data System. *AIAA Space Technology Conference and Exposition*, Albuquerque, NM, September 1999.

[8] E. M. Dashofy, A. Van der Hoek, R. N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *24the International Conference on Software Engineering*, Orlando, Florida, May 2002.

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

[10] M. M. Gorlick and R. R. Razouk. Using Weaves for Software Construction and Analysis. *13th International Conference on Software Engineering*, Austin, TX, May 1991.

[11] R. S. Hall, D. M. Heimbigner, and A. L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[12] B. Hayes–Roth et. al. A Domain-Specific Software Architecture for Adaptive Intelligent Systems. *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995.

[13] IONA Orbix/E Datasheet. *http://www.iona.com/whitepapers/orbix-e-DS.pdf*

[14] E. A. Lee. Embedded Software. *Advances in Computers* (Marvin V. Zelkowitz, ed.), Vol. 56, Academic Press, London, 2002.

[15] LIME. http://lime.sourceforge.net/

[16] J. Magee, J. Kramer. Dynamic structure in software architectures. *4th ACM SIGSOFT symposium on Foundations of software engineering*, San Francisco, CA, October 1996.

[17] C. Mascolo et. al. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Personal and Wireless Communications*, Kluwer, April 2002.

[18] N. Medvidovic, N. R. Mehta, M. Mikic-Rakic: A Family of Software Architecture Implementation Frameworks. *The Working IEEE/IFIP Conference on Software Architecture 2002*, Montreal, Canada, August 2002.

[19] N. Medvidovic, M. Mikic-Rakic, N. Mehta, S. Malek. Software Architectural Support for Handheld Computing. *IEEE Computer, special issue on handheld computing,* September 2003.

[20] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. *21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[21] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, January 2000.

[22] Microsoft .NET. http://www.microsoft.com/net/

[23] M. Mikic-Rakic and N. Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. *ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[24] M. Mikic-Rakic and N. Medvidovic. Architecture-Level Support for Software Component Deployment in Resource Constrained Environments. *Component Deployment, IFIP/ACM Working Conference*, Berlin, Germany, June 20-21, 2002.

[25] D.E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture**.** *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No.4, pages 40-52, October 1992.

[26] Project JXTA. http://www.jxta.org/

[27] M. Rakic, N. Medvidovic. Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. *2001 Symposium on Software Reusability (SSR 2001)*, Toronto, Canada, May 2001.

[28] R. V. Ommering. Building Product Populations with Software Components. *24th International Conference on Software Engineering*, Orlando, Florida, May 2002.

[29] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. *20th International Conference on Software Engineering,* Kyoto, Japan, April 1998.

[30] D. Schmidt. TAO. http://www.cs.wustl.edu/~schmidt/TAO.html

[31] B. Selic. Real-Time Object-Oriented Modeling (ROOM). *2nd IEEE Real-Time Technology and Applications Symposium,* June, 1996

[32] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. on Software Engineering*, April 1995.

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

[34] Sun Microsystems. JINI(TM) Network technology. http://wwws.sun.com/software/jini/

[35] W. Tracz. Domain-Specific Software Architecture Pedagogical Example. *ACM Software Engineering Notes*, July 1995.

[36] University of Southern California's "Software Engineering for Embedded Systems" class website: *http://sunset.usc.edu/classes/cs599_2002/index.html*

[37] S. Vestal. MetaH Programmer's Manual, Version 1.09. *Technical Report, Honeywell Technology Center,* April 1996.

[38] B. Werger. A Situated Approach to Scalable Control for Strongly Cooperative Robot Teams. *Ph.D. Thesis.* University of Southern California, May 2001.

[39] X2000/Mission Data System (MDS) project. *http://x2000.jpl.nasa.gov/nonflash/technology/mds.html*

[40] S. S. Yau and F. Karim, Context-Sensitive Middleware for Real-time Software in Ubiquitous Computing Environments. *Proceedings of the International Symposium on Object-oriented Real-Time Distributed Computing 2001*, Magdeburg, Germany.