

March 2010

Research Statement

Nenad Medvidovic

My research lies in the field of software engineering. A common theme, and long-term goal, of my research is *development and evolution of adaptable and dependable large-scale software systems*. Software practitioners have traditionally faced many problems with designing, implementing, deploying, and evolving software modules and/or systems. These problems are often the result of poor understanding of a system's overall architecture, unintended dependencies among its modules, decisions that are made too early or too late in the development process, over-reliance on specific implementation technologies, and so on. Traditional techniques that are intended to remedy these problems (e.g., separation of concerns or isolation of change) are only partially adequate in the case of development with pre-existing, large, multi-lingual components that originate from multiple sources.

The primary underlying hypothesis of my research is that an explicit architectural focus can remedy many of these difficulties and enable flexible construction and evolution of large software systems. I have therefore centered on *software architecture* as a key to developing techniques, tools, and methodologies for engineering flexible, large-scale software. Architecture comprises a system's principal design decisions and presents a set of high-level design views of the system. Architecture enables developers to abstract away the unnecessary details and focus on the system's building blocks (*components*), interactions (*connectors*), their structure (*configuration*), and key *properties*. A given system's architecture frequently adheres to one or more *architectural styles*, which are design guidelines shown in practice to result in systems with certain desired characteristics.

My work in this area began in the mid-1990s with the introduction of C2, an architectural style for graphical user interface-intensive systems. Over the years, the scope of my research has broadened significantly, to encompass a wide range of issues pertaining to software architectures and architecture-based development across a number of application domains.

To properly orient my research, over time I have also conducted several broad-based, foundational studies. Their aims were to establish the conceptual underpinnings and principles for different facets of software architecture-based development, which were invariably lacking in literature. These studies have resulted in a series of widely cited publications on *architecture modeling languages*, their *relationship to requirements and design modeling* (especially the Unified Modeling Language), interaction characteristics of software systems embodied in *software connectors*, and *architectural styles*. These studies culminated in a comprehensive software architecture textbook I co-authored, which was published in 2009.

In the remainder of this document will highlight the challenges faced and major results achieved by my research. While I use first person singular in my exposition for conven-

ience, as indicated in my *curriculum vitae*, a number of the described research results emerged from collaborative efforts.

Modeling — In addition to traditional architecture modeling concerns, which center around the key functional and interaction characteristics of a system, my work has resulted in several novel contributions to architecture modeling:

- development of a *type system* that aids flexible architectural evolution by supporting subtyping of different architectural concerns (specifically, name, interface, behavior, and implementation);
- expansion of architectural models to include the characteristics of *firmware* (hardware, operating system, and network) in support of system analysis, implementation, and deployment, especially in the arena of embedded and mobile systems;
- inclusion of the characteristics of application domains using, both, meta-models to guide the specification of *domain-specific modeling languages* and AI planning to model *system goals* independently of its architecture;
- accounting for the *partial and missing information* during architecture modeling via a suite of algorithms based on an existing formalism (MTS) that yield component models and controlled architectural refinement and abstraction;
- extension of architectural models to novel computing paradigms, in particular nature-inspired computing models, resulting in a first published demonstration that such models can not only be captured (Inverardi et al. already had showed this for CHAM, an architectural model inspired by chemical reactions) but also reified in actual distributed systems.

Analysis — From the inception, analysis has been a major focus of software architecture research. For the most part, the contribution of the early attempts has been in elevating the known analysis techniques to the architectural level. My work has focused on expanding the arsenal of analyses that are specifically relevant to and targeted at architectures:

- a suite of precise and scalable architecture-level system *reliability* estimation techniques, applicable at the level of individual components, specific system use-case scenarios, and entire systems;
- a tandem of analysis techniques for estimating the *energy consumption* induced by a given architectural style as well as the energy consumption of a particular system developed according to that style;
- a technique for achieving provable guarantees of computational and data *privacy* when deploying a system on a public, untrusted network;
- a technique for analyzing the *impact of changes* in an event-based architecture, targeted at aiding the maintainability and evolvability of event-based systems.

These specific analyses have been accompanied by two frameworks within which they (and other architectural analysis techniques) may be used:

- a *model interpreter framework* that leverages the domain-specific modeling languages discussed above to allow seamless inclusion of arbitrary quantifiable system properties into discrete event-based simulations;
- an *optimization framework* that suggests an improved deployment for a distributed system, given a set of (possibly conflicting) quality-of-service requirements elicited from system users.

Implementation — Architectural degradation – the discrepancy between a system’s implementation and its architectural model – is one of the most critical obstacles faced by engineers in large, complex, long-lived systems. Sophisticated architectural models and their properties established via equally sophisticated analyses are of little use unless they can be reified and maintained in the system’s implementation. Guided by this observation, a significant element of my work over the years has been on developing style-specific *architecture implementation frameworks*. This work has culminated in the development of an *architectural middleware* platform that supports the implementation of a system’s architecture in an arbitrary programming language, in a manner that ensures and preserves the key architectural properties in the system, including the (arbitrary) architectural style(s) to which the system should adhere. The architectural middleware is accompanied by support for automatically generating (partial) system implementations as well as event-based simulations from the architectural models. The concept of architectural middleware has been proven useful across a number of diverse domains, including grid computing, high-performance computing, wireless sensor networks, mobile robotics, and biologically-inspired distributed computing.

Deployment — As conceived, the architectural middleware allows the development of a set of higher-order services to aid system deployment (i.e., the placement of software components onto hardware hosts). These services are reified as meta-level components, which include support for

- *architectural reflection*, which enables the meta-level components to discover at runtime the relevant characteristics of the system’s underlying architecture;
- *service discovery*, which enables a system’s meta-level component on a given host to locate remote application-level components that provide services of interest;
- *runtime deployment analysis*, which allows an architecture to assess the current deployment and possibly suggest a new, improved deployment;
- *redemption*, which provides the support for each application-level component to be removed from the subsystem on its original host, properly packaged and sent across the network to its destination host, and installed on that host.

Adaptation — Change is endemic to software systems. My work has focused on leveraging software architecture in support of system adaptation both at design-time and runtime. The architectural type system, discussed above, is the engine behind *design-time adaptation*. In support of *runtime adaptation*, I have leveraged my work on architecture modeling, analysis, implementation, and deployment. These elements have been integrated in a runtime architecture-based adaptation framework, which was originally suggested in a paper presented at the 1998 International Conference on Software Engineering (ICSE). This adaptation framework relies on maintaining the mapping between a system’s architecture and its implementation, and mandates that the architecture be modified and those modifications analyzed before the implementation may be adapted. This framework has been widely adopted, and as part of my own research adapted and improved over time. This body of work was recognized in 2008, when the ICSE 1998 paper was given that conference’s *Most Influential Paper* award.

Recovery — Architectural degradation creeps into most software systems, eventually forcing engineers to recover a system’s architecture from its implementation artifacts. Architectural recovery is a vibrant research area, centered around static analysis to recover code-level elements and their relationships, which are then fed to clustering algorithms to identify higher-level structural elements. However, such techniques have been shown deficient in uncovering architectural intent. My work has tried to address that issue using a suite of techniques:

- Code typically contains hints that can help *identify the system’s components and connectors*. I have developed a technique that leverages a relatively small number of such hints (on the order of 10-12) to pinpoint correctly a large percentage (over 80% in several studies conducted to date) of architectural elements.
- Static analysis-based techniques are not effective at uncovering dependencies among system elements in a growing class of event-based systems. The *impact analysis* technique for event-based systems, discussed above, serves as a necessary starting point for aiding their architectural recovery.
- My recent work has begun identifying and classifying *architectural smells* – patterns in the organization of system elements that appear to violate certain “good design” practices (e.g., modularity, separation of concerns). Architectural smells are more difficult to identify automatically than the more widely studied code smells, but finding them can result in a comparatively greater benefit.
- My work has extensively leveraged architectural information that tends to remain reasonably stable – *architectural styles, reference architectures, and domain characteristics*. This information has been shown useful across diverse domains, most recently in recovering the architectures of a number of open-source grid technologies as well as those of dataflow-based scientific applications.

It is important to note that a significant element of my research has focused on producing results that solve real problems and have applicability outside the classroom and research laboratory. A number of my research results and tools have transitioned to third-party adopters. This is evidenced, both, by the range of organizations with which my publication co-authors are affiliated as well as the broad range of research funding sources.