Introducing MESSIA: A Methodology of Developing Software Architectures Supporting Implementation Independence[‡]

Ratko Orlandic
Department of Computer Science and Applied Math
Illinois Institute of Technology
10 West 31st Street, Chicago, IL 60616, U.S.A.
e-mail: ratko@charlie.cns.iit.edu

Abstract:
To sustain its rapid growth, software industry must embrace a distributed development paradigm in which complex systems are deployed by composing independently developed components. The glue to the development and integration of individual artifacts would be an elaborate software architecture. MESSIA is being developed as a methodology of software architectures that eliminate the possibility of architectural mismatch. This paper summarizes two important aspects of the methodology---its outcome and its design processes. The outcome of MESSIA is depicted by an abstract model of software architecture. The design processes are derived in a goal-directed fashion, from the knowledge of the organizational elements of the architectural model.

1. Introduction

Just as other areas of manufacturing, software industry must employ a highly distributed development paradigm in which complex systems would be deployed through compositions of individually developed artifacts. The glue to the development and integration of artifacts would be an elaborate component software architecture that would enable successful implementation and integration of individual artifacts. Typically, software architecture is viewed as set of *realms* [1], each of which provides implementation framework for a class of similar components.

A stumbling block for any compositional software construction is the problem of *architectural mismatch* [2]. The mismatch problems are caused by conflicting assumptions of independent vendors, primarily about the intended environment in which their components were to operate [2]. The actual source of these problems is the inadequacy of the architectural specification. The development of component systems hinges on our ability to deal with the problem of architectural mismatch effectively. Moreover, since it is not irrelevant whether the architecture incurs tens or tens of thousands instances of architectural mismatch, any success in reducing the number of such instances counts.

An effective approach to the problem of architectural mismatch requires a formal design methodology that can successfully guide the designers in developing mismatch-free software architectures. In our terminology, such an architecture is said to support *implementation independence* (*implementation autonomy*) [3]. We are in the process of developing such a methodology, which we named called MESSIA (**M**ethod of **E**volving **S**pecifications **S**upporting **I**mplementation **A**utonomy). The method relies on a theory of implementation independence [3,4] that provides an abstract characterization of architectural mismatch and derives the required conditions to achieve implementation independence. MESSIA is envisioned as a domain-specific method, targeting the domain of database management. However, in this paper, we ignore the related domain-specific issues.

In this paper, we summarize two important aspects of the methodology---its outcome and its design processes. The outcome of MESSIA is depicted by an abstract model of software architecture specifically

[‡] Work partially supported by DOE Grant DEFG02-95ER25254.

developed with the objective of implementation independence in mind. Section 2 introduces the model and discusses its basic organizational elements. Analyzing the structure of the architectural style, in Section 3, we derive the processes required to produce an architecture that supports implementation independence.

2. Structural Principles of Software Architecture

To tackle the problem of architectural mismatch, the designers must have an adequate model of software architecture that would allow them to express arbitrary implementation concerns. The model must also provide a minimal set of concepts allowing the designers to reason about the incompatibility of viable components. In addition, the model must provide an adequate set of architectural restrictions using which the designers can prevent the possibility of architectural mismatch.

In [4], we proposed a new model of software architecture. The organizing principle behind the model is that to reason about system implementation, it is necessary to understand the partition of the system into realms, the relevant implementation dilemmas of each realm, and the alternative solutions which may be used to resolve the dilemmas. By *implementation dilemma*, we mean any question concerning the implementation of a component that can have more than one answer. A *resolution* is one of the possible answers that, if changed, produces a system different from the original in some significant way. Since different high-level resolutions may lead to different kinds of lower-level dilemmas, which dilemmas will appear in the domain of a certain realm is partly determined by the adopted resolutions.

In the model, each realm is defined as a strict hierarchy of dilemmas and resolutions, called a *dilemma-resolution tree* (*DR-tree*) [4]. A DR-tree (e.g., the one illustrated in Figure 1) depicts the design decisions pertained to the definition of a realm. Each interior node of the hierarchy represents either an implementation dilemma or an allowed resolution, but a leaf is always a resolution option. The root of the tree is a dilemma that may be artificially introduced just to keep the hierarchy strict. A node representing a dilemma may have one or more children that denote alternative resolutions of the dilemma. In turn, an allowed resolution may lead to zero or more lower-level implementation dilemmas.

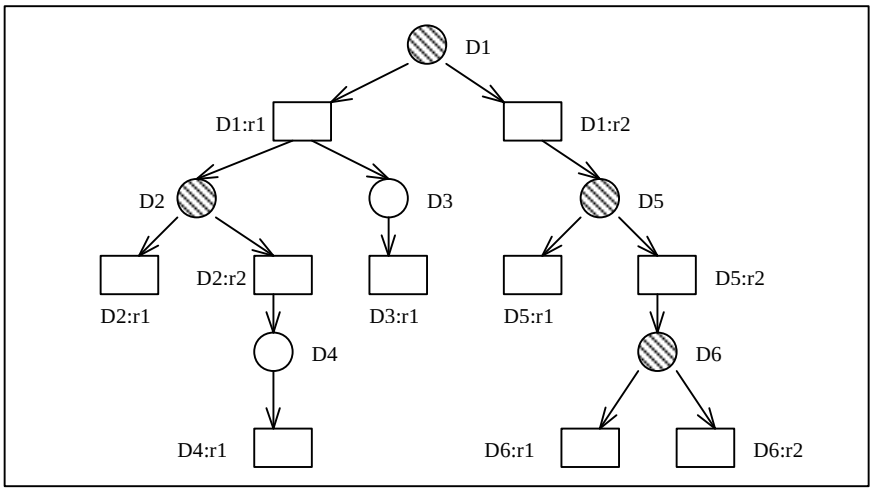

Figure 1. An abstract DR-tree of a certain realm.

The model defines a single concept using which the designers can reason about the incompatibility--*shared implementation dilemmas*. While the presence of shared dilemmas does not necessarily lead to architectural mismatch [4], their existence testifies to the possibility of such a mismatch. In Figure 1, the shared dilemmas of the realm are shaded. A dilemma appearing in more than one realm is regarded as shared. However, even a dilemma d_X that appears in the definition of only the realm X can be shared, provided that there is a resolution of a dilemma d_Y of another realm Y which "assumes" a particular way or

ways in which d_X must be resolved [4]. For uniformity, we require [4] that d_X be introduced in the realm *Y*, i.e. linked to an appropriate place in the DR-tree of *Y*. In [4], we have outlined a procedure for detecting the shared dilemmas on the DR-trees.

The theory of implementation independence [4] reveals that, to prevent architectural mismatch, each DR-tree must be deep enough to include all shared dilemmas of the corresponding realm. It also shows that the architectural mismatch can be prevented using only four types of architectural restrictions: *list a dilemma* as part of the definition of a realm, *list a resolution* option, *require a dilemma*, and *require a resolution* [4]. However, many types of shared dilemmas appear deep in the decision-making process, at levels that have been traditionally left up to the implementors. In practice, it makes sense to treat the permitted resolutions of these special classes of shared dilemmas separately, as global architectural assumptions. Consequently, the implementors would be free to decide in which implementation context to apply them, while the designers would have to worry about the contextual relationships of only a subset of dilemmas.

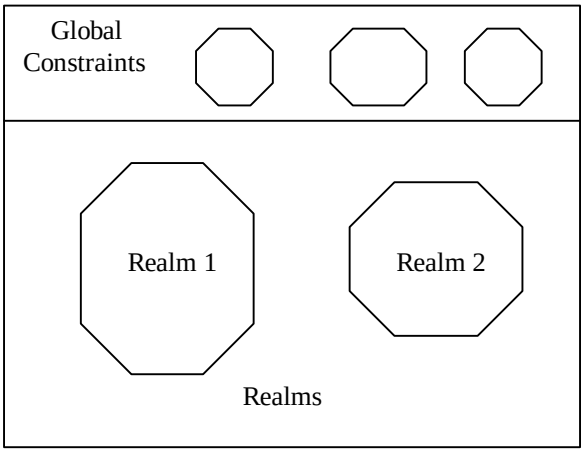

Figure 2. Structure of the architectural style.

The above reasoning leads to a conclusion that software architecture should be structured in at least two levels: global architectural assumptions and realms. This is illustrated in Figure 2. Each realm should be organized as a strict hierarchy of dilemmas and their resolutions (DR-tree) and the required dilemmas and should be distinguished from the optional ones. The global architectural constraints should be structured as a forest of DR-trees. The precise implementation context of "global" dilemmas is left up to the implementors.

3. Design Processes of MESSIA

The theory of implementation independence clearly shows that eliminating architectural mismatch using purely architectural is *not* an impossible task. However, in domains characterized by high degrees of coupling, this objective is extremely difficult to achieve. Part of the problem lies in the fact that shared dilemmas are highly elusive. Compounding the problem, in complex domains, there are many kinds of shared dilemmas that appear deep in the DR-trees and/or propagate throughout the system. In the process of detecting these types of shared dilemmas and determining their proper implementation context (in order to link them to the DR-trees), the task of the designers would look more like that of implementors.

Our work on MESSIA is motivated by a desire to find effective ways of dealing with the above and similar problems. In developing the method must, we must balance the need to simplify the design and reduce the size of the architectural specification, on one hand, with the necessity to eliminate any ambiguity in the specification, on the other.

The way an architectural specification is structured partly determines the approach to its construction. Therefore, MESSIA regards component software design as a process of refining the targeted

software architecture, beginning with a set of requirements. Referring back to Figure 2, one gets an idea of that steps are required to produce such an architecture---one needs to define the domain, decompose it into realms, identify the dilemmas that will be subject to global constraint, define the realms and global constraints, perform a dependency analysis and necessary consistency checks, and produce the final architectural specification. This implies the following broad processes of component design:

1) **Requirements specification**. To create any high quality software, the critical first step is to formulate the targeted domain in clear and consistent terms in the form of system requirements. The complexity of this process stems from the fact that system requirements are usually derived from some informal and often conflicting user requirements.
2) **Domain decoupling**. This is perhaps the most important and the most complex process of component design in which the designers must identify and factor out classes of similar components. In the process, the designers would perform a selection of base technology relying on a partial, rather than an exhaustive, analysis of implementation dilemmas [4].
3) **Detailed DR-tree construction**. A thorough treatment of all implementation dilemmas comes during the process of constructing detailed DR-trees of all realms previously identified. The process also determines the implementation context of the dilemmas and makes sure that the resolution sets of relevant dilemmas are restricted by listing the allowed resolution options.
4) **Exhaustive dependency analysis**. At this point, the designers would have enough detail about the realms and their domains to perform a thorough dependency analysis in order to identify the dilemmas shared across the realms.
5) **Cutoff analysis**. Following the dependency analysis, the designers would have a better insight into the types of shared dilemmas that should be subjects to global architectural constraint. Cutoff analysis is the process of extracting these global dilemmas and selecting their resolutions.
6) **Consistency checking and evaluation**. Next, the designers must verify whether a proper balance of all design objectives is achieved and whether the realms are locally and globally sound. In particular, the consistency rules derived by the theory of implementation independence [4] must be enforced.
7) **Final specification**. Finally, an architectural specification must be produced in accord to the selected architectural style and the architectural restrictions must be stated in accord to the consistency criteria.

There are numerous issues associated with the definition and the formalization of MESSIA. At present, our focus is on the key process of domain decoupling. The outcome of this process determines the degree of sharing between the individual realms and, consequently, the complexity of subsequent design, the size of the architectural specification, and the implementation complexity. In [5], we proposed a methodology of decoupling, targeting the domain of database management. The method proceeds in three phases: application-driven, service-driven, and implementation-driven decomposition. Each phase is designed to provide a framework for addressing one class of design objectives [5]. In the immediate future, we plan to continue our work on the formalization of software architecture and complete the definition of the methodology of DBMS decoupling.

References

1. D. Batory and S. O'Mally, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Trans. Software Engineering and Methodology*, **1**(4):355--398, 1992.
2. D. Garlan, R. Allen and J. Ockerbloom, "Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts," *Proc. 17th Int. Conf. on Software Engineering*, Seattle, WA, 179--185, 1995.
3. R. Orlandic, "A Theory of Implementation Independence: Architectural Specificity vs. Architectural Mismatch," *Proc. 10th Int. Conf. on Software Eng. and Knowledge Eng*. SEKE'98, San Francisco, CA, 140--145, 1998.
4. R. Orlandic, "A Formal Model of Reference Architecture and a Theory of Implementation Independence," technical document, 1998.

5. R. Orlandic, “Foundations of a Methodology of DBMS Decoupling for Evolutionary Component DBMS Design,” *Proc. Int. Database Engineering and Applications Symposium IDEAS '98*, Cardiff, Wales, UK, 178--187, 1998.