Round-Trip Software Engineering Using UML: From Architecture to Design and Back

Nenad Medvidovic[†] Alexander Egyed[†] David S. Rosenblum[††]

[†]Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
{neno,aegyed}@sunset.usc.edu

[††]Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
dsr@ics.uci.edu

Abstract

A key promise of software architecture research is that better software systems can result from modeling their important aspects throughout development. Choosing which system aspects to model and how to evaluate them are two decisions that frame software architecture research. Part of the software architecture community, primarily from academia, has focused on analytic evaluation of architectural descriptions. Another part of the community, primarily from industry, has chosen to model a wide range of issues that arise in software development, with a family of models that span and relate the issues. One problem that neither community has adequately addressed to date is round-trip software engineering: consistently refining a high-level model of a software system into a lower-level model (forward engineering) and abstracting a low-level model into a higher-level one (reverse engineering). This paper investigates the possibility of using the Unified Modeling Language (UML), an object-oriented design language, to that end. The paper assesses UML's suitability for modeling architectural concepts and provides a framework for identifying and resolving mismatches within and across different UML views, both at the same level of abstraction and across levels of abstraction. Finally, the paper briefly discusses our current tool support for round-trip software engineering.

1 Introduction

The basic promise of software architecture research is that better software systems can result from modeling their important aspects during, and especially early in the development. Choosing which aspects to model and how to evaluate them are two decisions that frame software architecture research.

Part of the software architecture research community has focused on analytic evaluation of architectural descriptions. A large number of architecture description languages (ADLs) has been proposed [10]. Each ADL embodies a particular approach to the specification and evolution of an architecture, with specialized modeling and analysis techniques that address specific system aspects in depth. Another part of the community has focused on modeling a wide range of issues that arise in software development, with a family of models that span and relate the issues of concern. However, by emphasizing breadth over depth, many problems and errors can potentially go undetected. One key cause is the lack of clear understanding of the relationship among the different models; two related problems are refining high-level models into lower-level models (forward engineering) and abstracting low-level models into higher-level ones (reverse engineering), depicted in Figure 1.

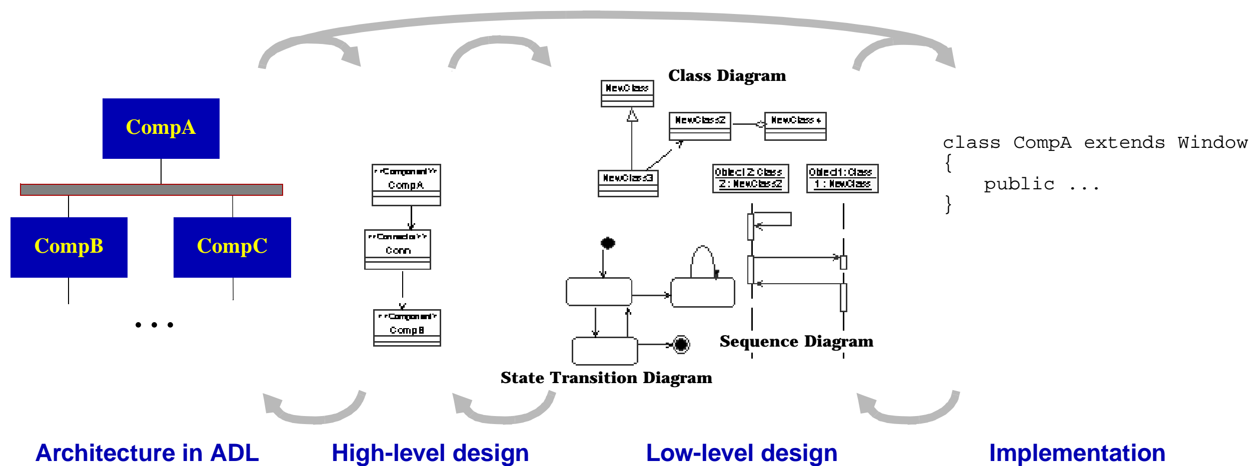

FIGURE 1. Round-trip software engineering. *Forward engineering*: a system model is refined from an architecture into designs at several levels of abstraction, and, eventually, an implementation. Certain system aspects may be implemented directly from an architectural description. *Reverse engineering*: a system's design and/or architecture is abstracted from its implementation. Modifications to a model at any level must be properly reflected in higher- and lower-level models.

Table 1 summarizes the two predominant approaches to addressing software architectures. Although the positions of the two communities are more complex than represented in the table, we believe that the table provides a useful, if simplified, overview of their relationship.

TABLE 1. Software Architecture Community Fragmentation

Academic Approach	Industrial Approach
focus on analytic evaluation of architectural models	focus on wide-range of development issues
individual models	families of models to span and relate the issues
rigorous modeling notations	practicality over rigor
powerful analysis techniques	architecture as the "big picture" in development
depth over breadth	breadth over depth
special-purpose solutions	general-purpose solutions

This issue paper investigates the possibility of using the Unified Modeling Language (UML) [11], an object-oriented design language, to span the two communities. UML is well suited for this because it provides a large, useful, and extensible set of predefined constructs, it is semi-formally defined, it has the potential for substantial tool support, and it is based on experience with mainstream development methods. The paper is based on a set of issues we have explored to date. The key aspects of our work have been:

- an assessment of UML's suitability for modeling architectural concepts provided by ADLs;
- a framework for identifying and resolving mismatches within and across different UML models, both at the same level of abstraction and across levels of abstraction; and
- tool support for integrating the above concepts into a round-trip engineering environment.

In this paper, we briefly present three possible approaches to using UML to model software architectures. We then discuss a view integration framework used to support automated validation of a modeled software system's integrity. Finally, we make some general observations on using UML to model software architectures, as well as ensuring consistent and complete refinement of architectures into designs and reverse-engineering of architectures from designs.

2 Using UML to Model Software Architectures

The four-layer metamodeling architecture of UML suggests three possible strategies for modeling software architectures in UML:

- use UML "as is,"
- constrain the UML meta model using UML's built-in extension mechanisms, and
- augment the UML meta model to directly support the needed architectural concepts.

We use a diagram that conceptually depicts UML's four-layer metamodeling architecture, shown in Figure 2, to illustrate the three approaches. Each approach has certain potential advantages and disadvantages for forward and reverse engineering, discussed below.

2.1 Strategy 1: Using UML "As Is"

The simplest strategy is to use the existing UML model to represent software architectures (Figure 2a) [7]. A major advantage of the approach is that it results in architectural models that are immediately understandable by any UML user and manipulable by UML-compliant tools. However, the approach would provide no means for explicitly representing the relationship between existing UML constructs and architectural concepts for which there is no direct UML counterpart (e.g., software connectors or architectural style rules). Rather, this relationship would have to be maintained implicitly by the software architect.

This approach would thus also present a considerable challenge in trying to reverse engineer a system's architecture from its UML model. None of the UML artifacts would contain any architectural information or explicitly represent architect's intentions. Instead, the architecture would have to be inferred from the design elements and their interactions. The repeatability of such a process is questionable: it is likely that different people would deem different elements of the UML model architecturally relevant, resulting in different (reverse engineered) architectures.

2.2 Strategy 2: Constraining UML

The space of software design situations and concerns for which UML is intended exceeds that of software architectures. Therefore, one possible approach to modeling architectures in UML is to constrain UML via *stereotypes* to address new concerns in software development. Conceptually, this approach can be represented using UML's metamodeling architecture from Figure 2b:

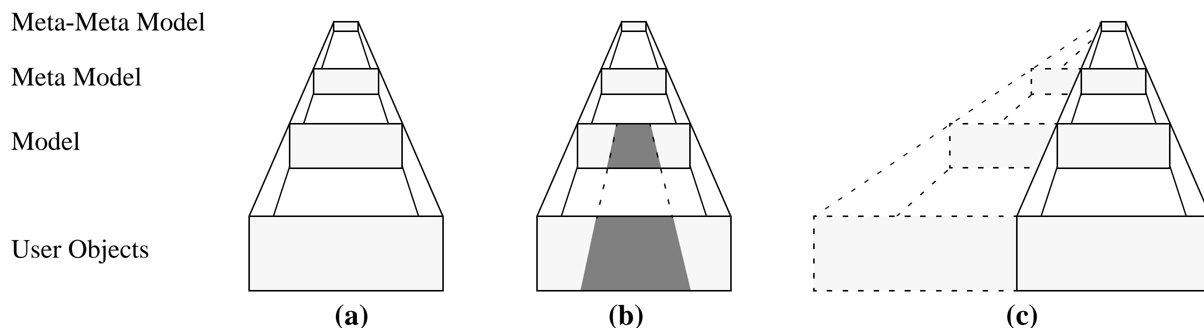

FIGURE 2. The four-layer metamodeling architecture of UML. The *meta-meta model* layer defines a language for specifying the meta model layer. The *meta model* layer, in turn, defines legal specifications in a given modeling language. For example, the UML meta model defines legal UML specifications. The *model* layer is used for modeling specific software systems, while the *user objects* layer corresponds to specific instances of a given model. ***(a)*** UML "as is" approach; ***(b)*** constrained UML approach (only subsets of the UML model and user object spaces are used); ***(c)*** augmented UML approach (the meta model, model, and user object spaces are extended).

only the relevant portion of the UML model is made available to the software architect. To date, we have applied this strategy to three ADLs [8,13]: C2 [9], Wright [2], and Rapide [6].

The major advantage of this approach is that it explicitly represents and enforces architectural constraints. Furthermore, an architecture specified in this manner would still be manipulable by standard UML tools and would be understandable by UML users (with some added effort in studying the stereotypes, partly expressed in OCL). Finally, the task of reverse engineering the architecture from a UML model with explicitly specified stereotypes would be greatly simplified.[1]

A disadvantage of the approach is that it may be difficult to fully and correctly specify the boundaries of the modeling space in Figure 2b. Additionally, as a practical concern, no tools that enforce OCL constraints in UML specifications currently exist. Finally, our extensive study of relating UML and ADLs using this strategy has shown that certain ADL features for modeling architectural semantics cannot be easily (or at all) represented in UML. An example is Rapide's event causality [8].

Another issue related to this strategy is the manner in which ADL-specific stereotypes are actually used within UML (see Figure 3). There are two possibilities:

- UML is used as the *primary* development notation, from which excursions are made to various ADLs (with the help of ADL-specific stereotypes) in order to exploit the existing ADL tool support;
- UML is used as the *only* development notation and ADL-specific stereotypes are accompanied by ADL-specific tools that have been modified to operate on UML specifications.

There are difficulties associated with both options. Using UML as the primary notation requires transformations both from a UML model to its ADL counterpart and from a possibly modified ADL model back to UML. This is a difficult task. To date, we have only shown how a UML model (extended via stereotypes) can be mapped to an ADL [13], but not vice versa. Using UML as the sole notation, on the other hand, requires modification, and perhaps reimplementation, of tool support that already exists for specific ADLs.

2.3 Strategy 3: Augmenting UML

One obvious, and therefore tempting, approach to using UML to support the needs of software architectures is to extend UML's meta model to directly support architectural concepts, as shown in Figure 2c. Extending the meta model helps to formally incorporate new modeling capabilities into UML. The potential benefit of such an extension is that it could fully capture every desired feature of every ADL. Furthermore, if features from the extended version are used in modeling a system's design, the task of reverse engineering the system's architecture from the design is greatly simplified.

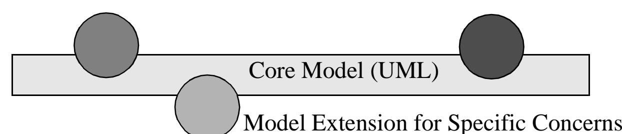

FIGURE 3. A UML model of a software system with ADL-specific extensions.

1. Clearly, having explicit stereotypes in a reverse engineered model requires that the engineer identify and designate important design/architectural elements abstracted from a lower-level model. This added task may require a slight change in the engineer's focus, but it does not fundamentally alter the nature of reverse software engineering.

However, such an extension to UML would not come without a price. The challenge of standardization is finding a language that is general enough to capture needed concepts without adding too much complexity, while such a modification would result in a notation that is overly complex. Moreover, unless the extensions were made part of the UML standard, they would be non-conforming, incompatible with UML-compliant tools, and potentially incomprehensible to architects.

3 Reconciling Architectural View Mismatches

A major emphasis in architecture-based software development is placed on identifying and reconciling mismatches within and among different views of a system (as enabled, e.g., by UML diagrams at different levels of abstraction). One facet of our work has been to investigate the ways of describing and identifying the causes of architectural mismatches in UML views. To this end, we have devised and applied a view integration framework, accompanied with a set of activities and techniques for identifying mismatches in an automatable fashion, described below [3].

This approach exploits redundancy between views: for instance, if view A contains information about view B, this information can be seen as a constraint on B. The view integration framework is used to enforce such constraints and, thereby, the consistency across the views. In addition to constraints and consistency rules, our view integration framework also defines *what* information can be exchanged across different views and *how* it can be exchanged. This is critical for automating the process of identifying and resolving inconsistencies.

The view integration framework is depicted in Figure 4. The System Model represents the (UML) model of the designed software system. In the course of forward software engineering, new information is added to the system model and existing views are updated; in the case of reverse engineering, information from existing views is abstracted to create new, higher-level views (View Synthesis). Whenever new information is added or a new view created, it must be validated against the system model to ensure its conceptual integrity (View Analysis). View Analysis involves the following major activities:

- *Mapping* identifies related pieces of information and thereby describes what information is overlapping. Mapping is often done manually via naming dictionaries or traceability matrices. A major part of our work has focused on automating this task by using patterns, shared interfaces, and inter-view dependency traces.

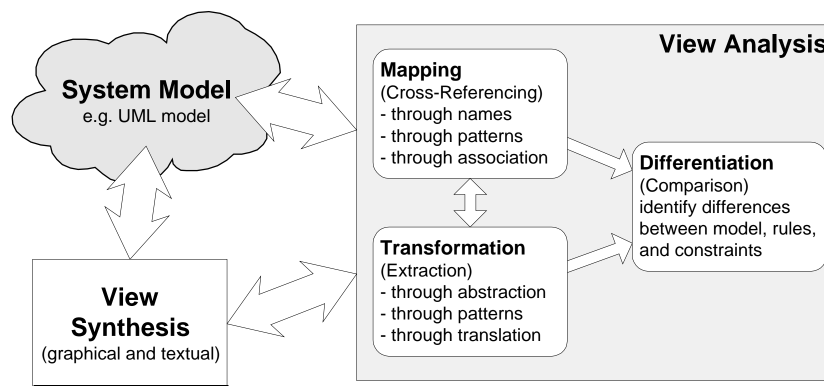

FIGURE 4. View integration framework.

- *Transformation* of model elements in order to simplify, i.e., generalize, a detailed view (abstraction) or exchange information between different types of views (translation).
- *Differentiation* traverses the model to identify potential mismatches within its elements. Potential mismatches can be automatically identified through the use of rules and constraints. Mismatch identification rules can frequently be complemented by mismatch resolution rules. Automated differentiation is strongly dependent on transformation and mapping.

To date, we have applied our view integration framework on several UML views: class and object diagrams, sequence diagrams, and statechart diagrams. We have also expanded the use of the framework beyond UML, to architectural styles (e.g., C2 [14], pipe-and-filter, layered, etc.) and design patterns.

A strong benefit of this framework is that it can be applied to both forward and reverse engineering. In forward engineering, the architect may start off by describing the architecture, followed by the design (and, subsequently, the implementation). Our view integration framework may then be applied to ensure that the design is consistent internally as well as with higher-level abstractions (including the architecture). Thus, the *first step* includes the creation of views at various levels, whereas the *second step* validates the conceptual integrity of those views.

In reverse engineering, the architect follows essentially the same procedure. Obviously, a distinction is in step one where the architecture is now created from the design; however, the validation process (step two) remains the same. When ensuring the integrity of two models, it does not matter which one is created first: inter-view consistency and completeness can be validated either way.

As outlined in our integration framework, consistency between views at various levels of abstraction is enabled by transforming lower-level views into higher-level ones. Thus, our framework has the potential to further support reverse-engineering by semi-automatically generating higher-level abstractions from lower-level models. This is achieved by using the transformation techniques summarized above. At that point, step two of our reverse engineering process is applied to ensure that subsequent changes to the derived abstraction (e.g., the architecture) remain consistent with the initial model (e.g., the design).

4 Observations

Our effort to date has furthered our understanding of UML's suitability for supporting architecture-based, round-trip software engineering. We have gained valuable insights on which we intend to base our future work. These insights are discussed below.

Software Modeling Philosophies. Neither UML nor ADLs constrain the choice of implementation language or require that any two components be implemented in the same language or thread of control. ADLs or styles may assume particular communication protocols and UML typically supports such restrictions. The behavior of architectural constructs (components, connectors, communication ports, and so forth) can usually be modeled with UML's sequence, collaboration, statechart, and activity diagrams. Existing ADLs are usually able to support only a subset of these kinds of semantic models.

Assumptions. Like any notation, UML embodies certain assumptions about its intended usage. Software "architecting," in the sense it is often used in the architecture community (by employing conceptual components, connectors, and their configurations, exploiting rules of specific architec-

tural styles, and modeling local and global architectural behavior and constraints), was not an intended use of UML. A software architect may thus find that the support for the desired architectural constructs found in UML only partially satisfies his/her needs.

Problem Domain Modeling. UML provides extensive support for modeling a problem domain. Architectural models described in ADLs, however, often hide much of the information present in a domain model. Modeling all the relevant information early in the development lifecycle is crucial to the success of a software project. Therefore, a domain model should be considered a separate and useful architectural perspective.

Architectural Abstractions. Some concepts of software architectures are very different from those of UML. For example, connectors are first-class entities in many ADLs. We have demonstrated that the functionality of a connector can typically be abstracted by a class or component. However, connectors may have properties that are not directly supported by a UML class. The underlying problem is even deeper. Although UML may provide modeling power equivalent to or surpassing that of an ADL, the abstractions it provides may not match an architect's mental model of the system as faithfully as the architect's ADL of choice. If the primary purpose of a language is to provide a vehicle of expression that matches the intuitions and practices of users, then that language should aspire to reflect those intentions and practices. We believe this to be a key issue: a given language (e.g., UML) offers a set of abstractions that an architect uses as design tools; if certain abstractions (e.g., components and connectors) are buried in others (e.g., classes), the architect's job is made more (and unnecessarily) difficult; separating components from connectors, raising them both to visibility as top-level abstractions, and endowing them with certain features and limitations also raises them in the consciousness of the designer.

Architectural Styles. Architecture is the appropriate level of abstraction at which rules of a compositional style (i.e., an architectural style) can be exploited and should be elaborated. Doing so results in a set of heuristics that, if followed, will guarantee a resulting system certain desirable properties. Standard UML provides no support for architectural styles; the rules of different styles somehow have to be built into UML. We have done so by using stereotypes. One potential problem with this approach, as already discussed, is ensuring that style rules are correctly and completely captured in UML.

Architectural Views. ADLs typically support modeling of a limited number of architectural views, but ensure their full consistency and interchangeability. UML, on the other hand, allows designers to model a system from many perspectives, but does not provide mechanisms for ensuring their consistency. The presence of multiple, possibly inconsistent views in a UML model is very likely to make the task of reverse engineering more challenging. Both UML and ADLs can therefore benefit from techniques for view mismatch identification and reconciliation. One such set of techniques was discussed in Section 3.

5 Current Status and Future Work

We intend to expand this work in several directions, including providing tool support for using UML in architecture modeling, maintaining traceability and consistency between architectural and design decisions, and combining the existing implementation generation and reverse engineering capabilities for ADLs and UML. We also intend to draw upon our experience to date to

suggest specific extensions needed in the UML meta model to better support software architectures.

We have already begun to address several of these issues. We have developed an initial integration of DRADEL [9], an environment for C2 style architecture-based development, with Rational Rose [12], an environment for software design and implementation with UML [1]. The integration enables automated mapping from an architecture described in C2's ADL into UML using both Strategies 1 and 2. Currently, this mapping is uni-directional and the UML model is consistent with respect to the architecture only initially; any subsequent refinements of the UML model may violate architectural decisions. Also, as additional views are introduced into the design (e.g., activity and deployment diagrams), their consistency with the existing views (e.g., state and class diagrams) must be ensured. To this end, we are beginning to develop a set of techniques and associated tool support to ensure full integration of views in UML [4,5]. The resulting tool, UML/Analyzer, provides automated support for forward and reverse engineering using UML class diagrams. In the future, this support will be extended to other kinds of UML diagrams and to architectures modeled in DRADEL.

6 References

[1] M. Abi-Antoun and N. Medvidovic. Enabling the Refinement of a Software Architecture into a Design. Submitted for publication, 1999.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[3] A. Egyed. Automating Architectural View Integration in UML. Technical Report USCCSE-99-511, Center for Software Engineering, University of Southern California, Los Angeles, CA, 1999.

[4] A. Egyed and P. Kruchten. Rose/Architect: A Tool to Visualize Architecture. *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS-32)*, January 1999.

[5] A. Egyed and N. Medvidovic. Extending Architectural Representation in UML with View Integration. Submitted for publication, 1999.

[6] D.C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering* 21(9), September 1995.

[7] N. Medvidovic and D.S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. In *Proceedings of the First IFIP Working Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 1999.

[8] N. Medvidovic, D.S. Rosenblum, J.E. Robbins, and D.F. Redmiles. Modeling Software Architectures in the Unified Modeling Language. Submitted for publication, 1999

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

[10] N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. In *IEEE Transactions on Software Engineering*, to appear.

[11] Object Management Group. *OMG UML Specification Version 1.3 R9 Draft*. January 1999. Accessed at Web site http://uml.shl.com/.

[12] Rational Software Corporation. *Rational Rose 98: Using Rational Rose*.

[13] J.E. Robbins, N. Medvidovic, D.F. Redmiles, and D.S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

[14] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead, Jr., J.E. Robbins, K.A. Nies, P. Oreizy and D.L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.