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. 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 how the different models are related.

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

<table>
<thead>
<tr>
<th>Academic Approach</th>
<th>Industrial Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>focus on analytic evaluation of architectural models</td>
<td>focus on wide-range of development issues</td>
</tr>
<tr>
<td>individual models</td>
<td>families of models to span and relate the issues</td>
</tr>
<tr>
<td>rigorous modeling notations</td>
<td>practicality over rigor</td>
</tr>
<tr>
<td>powerful analysis techniques</td>
<td>architecture as the “big picture” in development</td>
</tr>
<tr>
<td>depth over breadth</td>
<td>breadth over depth</td>
</tr>
<tr>
<td>special-purpose solutions</td>
<td>general-purpose solutions</td>
</tr>
</tbody>
</table>

This issue paper investigates the possibility of using the UML 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. Two key aspects of our work have been:
- an assessment of UML’s suitability for modeling architectural concepts provided by ADLs, and

---

1. This issue paper has been written in collaboration with Marwan Abi Antoun, Alexander Egyed, and Nikunj Mehta (University of Southern California), and David Redmiles, Jason Robbins, and David Rosenblum (University of California, Irvine).
• a framework for identifying and resolving mismatches both within and across different UML models of a single system.

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.

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 1, to illustrate the three approaches. Each approach has certain potential advantages and disadvantages, 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 1a). 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.

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. Conceptu-
ally, this approach can be represented using UML’s metamodeling architecture from Figure 1b: 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: C2, Wright, and Rapide.

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). A disadvantage of the approach is that it may be difficult to fully and correctly specify the boundaries of the modeling space in Figure 1b. 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.

Another issue related to this strategy is the manner in which ADL-specific stereotypes are actually used within UML (see Figure 2). 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 mappings 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, 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 1c. 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. However, 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 and may be incompatible with UML-compliant tools.
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 different UML diagrams). 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.

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 3. The System Model represents the (UML) model of the designed software system. In the course of software development, new information is added to the system model and existing views updated (View Synthesis). Whenever new information is added, 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.
- **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 are expanding the use of the framework beyond UML, to architectural styles (e.g., C2, pipe-and-filter, layered, etc.) and design patterns.

FIGURE 3. View integration framework.
4 General Observations

Our effort to date has furthered our understanding of UML’s suitability for supporting architecture-based software development. 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 architectural 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 prob-
lem 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. Both UML and ADLs can therefore benefit from techniques for view mismatch identification and reconciliation.

5 References

On ADLs:

   http://sunset.usc.edu/~neno/papers/TSE-ADL.pdf

On the importance and use of first-class connectors:


On using UML to support architecture-based software development:

   http://sunset.usc.edu/~neno/papers/ICSE98-UML.pdf

   http://sunset.usc.edu/~neno/papers/WICSA1.pdf

On view mismatch:

   http://sunset.usc.edu/TechRpts/Papers/usccse98-514/usccse98-514.pdf

   http://sunset.usc.edu/TechRpts/Papers/usccse99-511/usccse99-511.pdf