Consistent Architectural Refinement and Evolution using the Unified Modeling Language

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 231
Los Angeles, CA 90292, USA
aegyed@acm.org

Nenad Medvidovic
University of Southern California
941 W. 37th Place, Suite 327
Los Angeles, CA 90089-0781, USA
neno@usc.edu

ABSTRACT
Architecture Description Languages (ADLs) comprise a sizeable set of modeling techniques that are aimed at bridging the gap between requirements engineering and low-level design and code. ADLs excel in their ability to model high-level functional and non-functional aspects of software systems and have demonstrated increasing support for trade-off analyses (i.e., requirements feasibility) and simulation. On the downside, ADLs are highly specialized and tend to rely on abstract notions such as roles and responsibilities. This creates problems when it comes to refining software artifacts into platform-specific programming constructs and combining solutions derived via different ADLs. Over the last years, we have devised mechanisms for transforming architecture models into implementation code by leveraging the Unified Modeling Language (UML). This paper presents an overview of our approach which is accompanied by extensive tool support.

Keywords
Refinement, Transformation, Consistency, Evolution, ADL, UML

1. INTRODUCTION
The basic promise of software architectures is that better software systems can result from modeling their important aspects during, and especially early in the development. Architecture-based development has received a lot of attention in academia and industry as evidenced by numerous languages, techniques, and tools [13]. A part of the software architecture research community has focused on analytic evaluation of architectural descriptions, resulting in large numbers of architecture description languages (ADLs). 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. On the downside, highly specialized architecture description languages (ADLs) only address specific modeling and analysis issues (e.g., architectural structure), rarely focusing on the broader development picture.

With the emergence and widespread acceptance of the Unified Modeling Language (UML) [2], it has become obvious that another parts of the software engineering community is interested in broader modeling languages that address a wide range of concerns that arise in software development. Those modeling languages span families of models and relate them under a common umbrella (i.e., meta model). The generic nature of UML makes is more suitable for addressing general software modeling problems. However, by emphasizing breadth over depth allows many problems and errors to go potentially undetected. One such set of problems deals with consistency across related UML models [4].

In our work on combining general-purpose modeling (i.e., UML) with specific-purpose modeling (i.e., ADLs), we have found that the two approaches complement one another [10]. Thus, for instance, UML emphasizes modeling practicality and breadth, while ADLs emphasize rigor and depth. Both these perspectives are needed to address the broad spectrum of rapidly changing situations that arise in software development.

This paper discusses our attempts at unifying UML and ADLs that also aid in the automated refinement of architectures to implementation and in the baselining of different architectural modeling languages. Towards refinement, we use UML as a bridge to transform architectural elements into design elements that are more easily refined into code. Towards baselining, we use UML as a common foundation for multiple ADLs, to be used for data exchange or consistency checking. Thus, we discuss here two issues – architectural refinement and ADL baselining to ensure model consistency – in the remained of the paper.

2. ARCHITECTURAL REFINEMENT
Architectures focus on abstract concepts such as software components roles, responsibilities, and interaction protocols, whereas source code emphasizes concrete concepts such as iterations, variables, and method calls. We have found that design

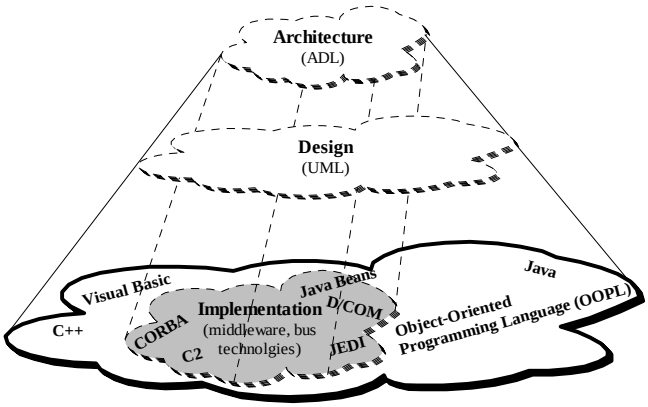

Figure 1. From Architecture to Implementation [1]

Proceedings of the 1st Workshop on Describing Software Architecture with UML, co-located with ICSE 2001, Toronto, Canada, May 2001, pp. 83-87.

models such as those enabled by the UML can serve as a useful bridge in refining architectural models to code. Specifically, UML allows one to represent architectural concepts with design-level constructs, which are conceptually closer to the implementation. Thus, the design supplements the architecture by modeling added details on how roles, responsibilities, and interaction protocols may be realized. The design typically also extends the architecture by further subdividing architectural components into smaller elements and providing additional details for them as well.

For our purposes, the starting point of refinement is an architecture composed of coarse-grained components and connectors, and their configurations [1]. The architecture adheres to some architectural style (e.g., client-server, pipe-and-filter, layered) and its representation may be formalized using an ADL. That architecture is refined into a design, and eventually, an implementation. Given that architectures are intended to describe systems at a high-level of abstraction, directly refining an architectural model into a design or implementation may not be possible. One reason is that the design space rapidly expands with the decrease in abstraction levels, as shown in Figure 1. A solution to this problem is to bound the target (implementation) space by employing specific middleware technologies. A more general refinement approach includes design as an intermediate step. During design, constructs such as classes with attributes, operations, and associations, instances of objects collaborating in a scenario, and so forth, are identified. These are more effectively expressed in a notation such as UML than in an ADL. However, various problems might arise. First, the design may no longer be faithful to the rules of the selected architectural style. Second, maintaining traceability is inherently difficult, because of the possible many-to-many mappings from the elements in the problem domain to the elements in the solution space [5]: a given element from one space can map to zero, one, or more elements in the "lower level" space. Third, some architectural elements, such as connectors, are given first class status in architectures [15], but may not have direct design or implementation counterparts. Typically, connectors are "designed away" into various class associations and object interactions or are "coded away" into programming language statements distributed across different components. In [10], we discuss various refinement strategies that can be taken in refining ADLs to UML:

- Strategy #1 consists of using standard UML constructs to simulate modeling architectural concerns as would be done in an ADL [12].
- Strategy #2 consists of using UML's built-in extension mechanisms (stereotypes and tagged values) [2] and the Object Constraint Language (OCL) [17] to constrain the semantics of meta-classes to those of ADL constructs [9].
- Strategy #3 consists of augmenting the UML meta-model to directly support architectural concerns. Although this is a potentially effective approach, it would result in a notation that is incompatible with standard UML. Since one of our goals for

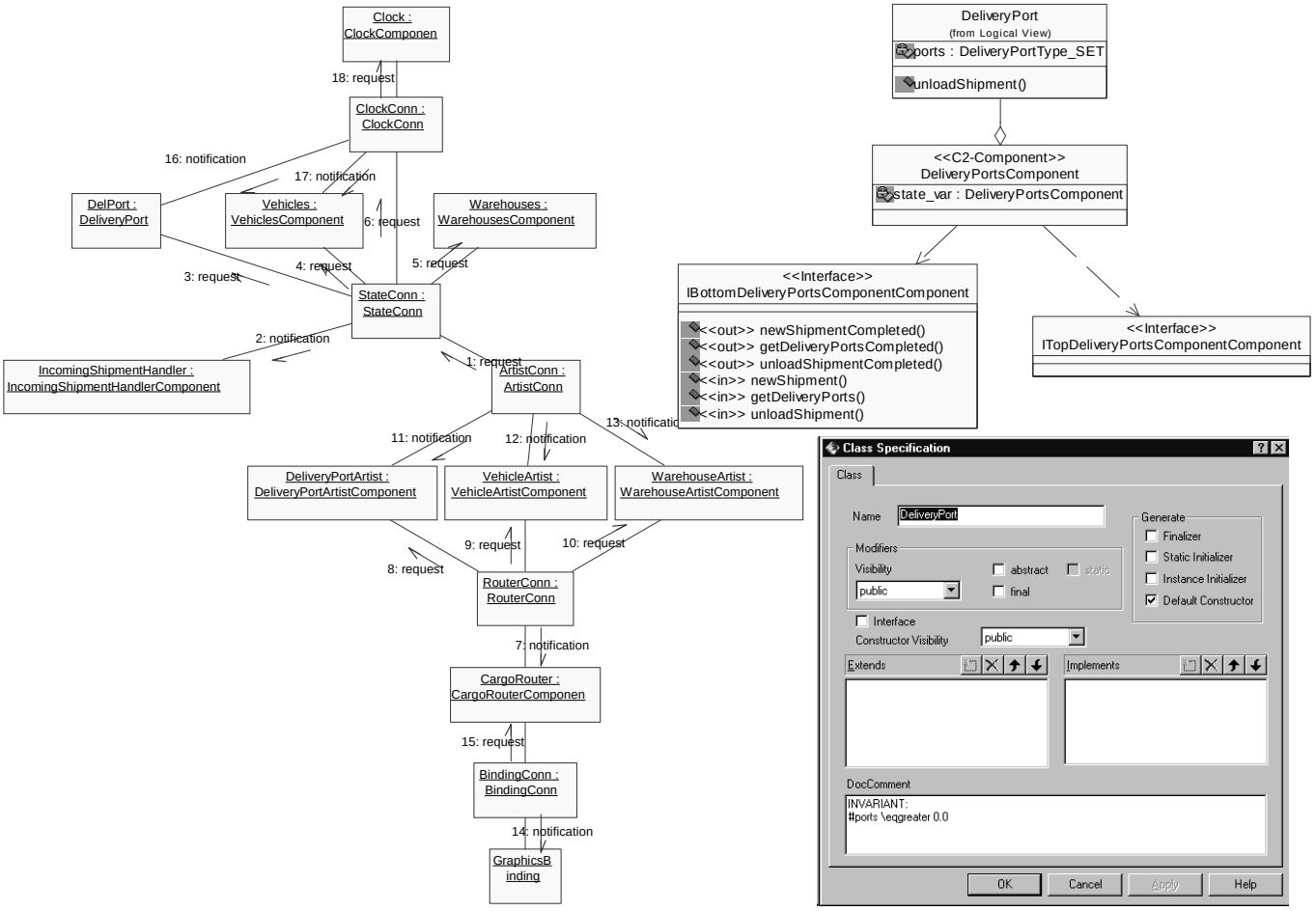

Figure 2. Partial UML model of CargoRouter architecture (using Rational Rose™)

this work is conformance with standard UML and corresponding tools, we do not pursue this strategy currently.

It is out of the scope of this paper to discuss refinement strategies in detail; in [1] we demonstrated all three in the context of an example. Although, all three refinement strategies have respective advantages and disadvantages, we have found Strategy #2 to be the least intrusive and most expressive option. In [12] we describe this strategy and discuss how we use UML stereotypes to represent modeling constructs of several ADLs in UML. Specifically, we created a development environment called SAAGE which automates this transformation in context of the C2SADEL specification language [16].

The UML specification resulting from this transformation is stored as a Rational Rose™ model [14]. A portion of the Rose model for an application architecture describing a cargo router is depicted in Figure 2: it depicts a part of the C2 architecture as a UML collaboration diagram (left) and the attributes of and class diagram corresponding to the *DeliveryPort* component (right).

The details of how our refinement approach are outside the scope of this paper, however, it must be noted that the automatically generated Rose model is consistent with the C2 architectural model and is used as the basis for further, possibly manual refinement. For instance, we could now use Rational Rose's code generation capabiltiy to generate skeleton code.

3. CONSISTENCY

The refinement mechanism discussed above is adequate for representing architectures in UML. The only major limitation is that refinement comes up short in fully integrating UML views with each other since both refinement and abstraction may evolve independently of one another thereafter: Although UML and its meta-model define notational and semantic aspects of individual views in detail, inter-view relationships are not captured in sufficient detail. Without this information, the (UML) model is nothing more than a collection of loosely coupled (or even unrelated) views. The lack of view integration extends beyond existing UML views [4,5] to non-UML views represented in UML (e.g. ADLs) [7]. For instance, if architectural models are refined via UML then we need to make sure that those refinements remain consistent with the original architecture. Furthermore, if multiple ADLs are represented in UML (as proposed in Section 2) then we also need to also make sure that those different architectural views are consistent with one another.

Take, for instance, the separately created object diagram (Figure 2 right) which depicts a refinement of the architecture from Figure 1 (note that Figure 2 replicates a part of the architecture from Figure 1 for ease of comparison). By themselves, views do not constrain one another unless constraints are somehow defined. In Figure 2, we thus depict various mappings of relationships between the two views; mappings (traces) from UML objects to C2 components/connectors are shown with dotted lines. Thus, we see that the objects *theStorage*, *RefrigeratedStorage*, and *aWarehouse* belong conceptually to the architecture component *Warehoue* (see Figure 2 left) and we can see similar mappings for other elements. Basically, C2 components and connectors may be seen as the interfaces for compact, self-sustaining sections of the implementation. Since C2 elements (components and connectors) are often coarse grain, it is reasonable to assume that a collection of objects is needed to implement a single C2 element.

Because of the fact that a C2 element is a black box, nothing can be said about how objects part of a single element are supposed to interact. However, the interactions of objects belonging to different C2 elements are constrained by the C2 style. For instance, if the design-object *aWarehouse* corresponds to the C2 component *Warehouse*, and if the design-object *aPort* corresponds to the C2 component *Port*, then the interaction between the two groups of objects needs to be consistent with the interaction of the corresponding C2 architectural elements. For instance, in C2, components at the same "level" are not allowed to communicate (i.e., Warehouse cannot communicate with DelPort). In the design, we find that this is violated since *aSurplus* is part of *aWarehouse* and *aPort* is part of *aSurplus*. Transitively, we can thus infer that *aPort* is part of *aWarehouse* (this reasoning is discussed in more detail in [6]). Likewise, we find that in the design there is no connection between *aClock* and other design objects, thus again violating the architecture which specifies that the component *Clock* may interact with components *DelPort* and *Vehicle* (via a connector).

Our approach to automated consistency checking, called *IViTA* (inter View Transformation and Analysis), can detect such

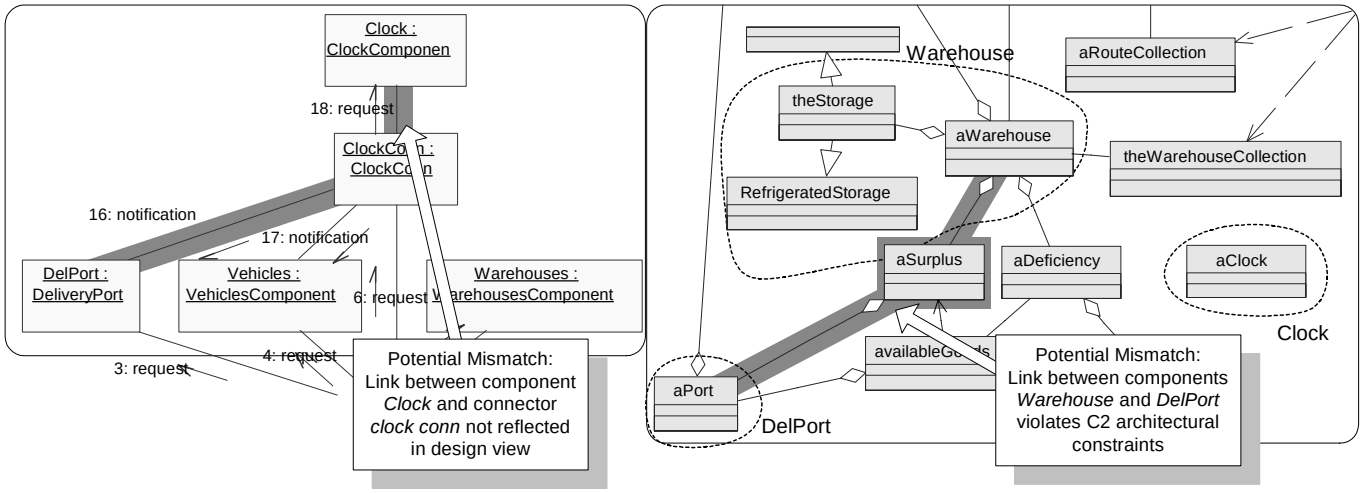

Figure 2. C2 and corresponding Class View plus Mismatches

inconsistencies. *IViTA* is a conceptual framework for transformation-based consistency checking [5]. In context of UML and C2, *IViTA* has been automated in a tool called UML/Analyzer [4]. *IViTA* combines consistent transformation and consistency comparison. Consistent transformation ensures consistency via well-defined transformation steps where source models are transformed into target models in a manner that guarantees consistency. Consistency comparison, on the other hand, detects inconsistencies via well-defined comparison steps. Our approach is conceptually similar to consistency checking approaches like VisualSpecs [3] or JViews [8], both of which use transformation to convert graphical models into either a formal language (VisualSpecs) or a data repository (JViews) in which they perform consistency analyses. There is, however, one major distinguishing factor in that we do not belief that either a single formal language or a single meta-model can be found that represents the vast variety of modeling languages available today. Even in cases like the Unified Modeling Language (UML), where a single meta-model is available, comparing model elements in that language is still non-trivial and can often not be done in a simple manner.

We thus use transformation to convert source models into the types of target models we wish to compare to. For instance, if we would like to compare the class diagram in Figure 2 with the architecture diagram in Figure 1, our approach would either reverse engineer the class diagram to yield an architecture followed by comparing both architectures; or our approach would generate another class diagram out of the architecture followed by comparing both versions of the class diagrams. It is our emphasis to bring models closer to one another to simplify comparison. Abstraction and refinement is only one dimension of transformation. In our investigation on how to bridge models, we have identified other transformation methods as outlined in [11].

4. CONCLUSION

This paper discussed highlights of our research in architectural modeling, refinement, and consistency checking. As part of refinement, we use UML as an intermediate language to simplify the mapping between architecture and code. For consistency checking, we use UML's ability to provide a common modeling baseline to compare those models. To date, we have automated many of the concepts discussed in this paper and created two tool suites: SAAGE to enable architectural modeling and refinement; and UML/Analyzer (also integrated with Rose) to enable automated transformation and consistency checking. It is our long-term vision to further extend the automation support to other parts of the software life cycle and other architecture description languages.

5. REFERENCES

1. Abi-Antoun, M. and Medvidovic, N.: "Enabling the Refinement of a Software Architecture into a Design," Proceedings of the 2nd International Conference on the Unified Modeling Language (UML), October 1999.
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison Wesley, 1999.
3. Cheng, B. H. C., Wang, E. Y., and Bourdeau, R. H.: "A Graphical Environment for Formally Developing Object-Oriented Software," Proceedings of IEEE International Conference on Tools with AI, November 1994.
4. Egyed, A.: "Heterogeneous View Integration and its Automation," PhD Dissertation, University of Southern California, Los Angeles, CA, August 2000.
5. Egyed, A.: "Taming Ambiguity to Overcome the Model Consistency Barrier," submitted to European Conference on Software Engineering and Foundations of Software Engineering (ESEC/FSE), Vienna, Austria, September 2001.
6. Egyed, A. and Kruchten, P.: "Rose/Architect: a tool to visualize architecture," Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS), January 1999.
7. Egyed, A. and Medvidovic, N.: "Extending Architectural Representation with View Integration," Proceedings of the 2nd International Conference on the Unified Modeling Language (UML), October 1999.
8. Grundy J., Hosking J., and Mugridge R.: Inconsistency Management for Multiple-View Software Development Environments. IEEE Transactions on Software Engineering (TSE) 24(11), 1998.
9. Medvidovic, N., Oreizy, P., and Taylor, R. N.: "Reuse of Off-the-Shelf Components in C2-Style Architectures," Proceedings of the 1997 Symposium on Software Reuseability (SRR'97) and Proceedings of the 1997 International Conference on Software Engineering (ICSE'97), Boston, MA, May 1997.
10. Medvidovic, N., Egyed, A., and Rosenblum, D.: "Round-Trip Software Engineering Using UML: From Architecture to Design and Back," Proceedings of the 2nd Workshop on Object-Oriented Reengineering (WOOR) , September 1999, pp.1-8.
11. Medvidovic, N., Gruenbacher, P., Egyed, A., and Boehm, B.: "Software Lifecycle Connectors: Bridging Models across the Lifecycle," accepted at the International Conference on Software Engineering and Knowledge Engineering (SEKE 2001), June 2001.
12. Medvidovic, N. and Rosenblum, D. S.: "Assessing the Suitability of a Standard Design Method for Modeling Software Architectures," Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), February 1999 , pp.161-182.
13. Medvidovic N. and Taylor R. N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26(1), 2000, 70-93.
14. Rational Corporation, Rational Rose, http://www.rational.com/.
15. Shaw, M.: "Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status," Workshop on Studies of Software Design, 1993.
16. Taylor R. N., Medvidovic N., Anderson K. N., Whitehead E. J. Jr., Robbins J. E., Nies K. A., Oreizy P., and Dubrow D. L.: A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering 22(6), 1996, 390-406.
17. Warmer, J. and Kleppe, A.: The Object Constraint Language. Reading, MA, Addison Wesley, 1999.