Published in the Proceedings of 4th International Software Architecture Workshop (ISAW) co-located with ICSE 2000, Limerick, Ireland, June 2000

Architectural Integration and Evolution in a Model World

Alexander Egyed
University of Southern California
Computer Science Department
941 W. 37th Place, SAL 327
Los Angeles, CA 90089-0781
aegyed@sunset.usc.edu

Rich Hilliard
Integrated Systems and
Internet Solutions, Inc.
150 Baker Avenue Extension
Concord, MA 01742, USA
rh@isis2000.com

ABSTRACT
Architectural Description Languages (ADLs) fall into the narrow category of frequently using only one fixed representation scheme. Over the past years, it has become more obvious that no single such ADL is adequate in addressing a large number of stakeholder concerns. This paper, therefore, discusses the need and challenges of multi-view development with ADLs and introduces a decorative stance in handling view integration issues and scalability concerns related to consistency checking.

Keywords
Model-based development, view integration, consistency,

1 INTRODUCTION

Boehm said that "your project will succeed if and only if you make winners out of all critical stakeholders." [2] This notion, coined in requirements engineering, implies that it is vital to identify critical stakeholders, capture their concerns and goals, and resolve conflicts between them to ensure that the developed software system meets everyone's expectations. By stakeholder we mean an individual or a group that shares concerns or interests in the system (e.g. developers, users, customers, etc.). Software architecting and design can also utilize these notions, so it is not surprising that recent standards such as the IEEE *Recommended Practice for Architectural Description* [8] (P1471) advocate using architectural views to address stakeholder concerns. Concerns can be of different origins: (1) goals to reflect the wishes and expectations of the stakeholders; and (2) conflicts to reflect clashes between these goals. To address development and evolution concerns, the P1471 standard suggests the use of views, following a widespread practice in software/systems modeling. Views deal with concerns in these ways:

Published in Proceedings of 4th International Workshop on Software Architecture (ISAW-4), Limerick, Ireland, June 2000 (co-located with ICSE 2000)

- Views can be used to separate concerns and reduce their overall complexity,
- Views can be used to describe and analyze the feasibility of stakeholder goals, and
- Views can be used to identify and resolve conflicts by suggesting options.

Since views address concerns (during development and evolution), modeling with views needs to handle issues such as feasibility, security, maintainability, performance, reliability, cost, schedule, interoperability and so forth. The focus of this paper is geared towards architectural views and since architectural modeling goes hand in hand with requirements modeling, it follows that architectural modeling could be used to generally validate concerns on a higher level (as compared to identifying problems on the source code level resulting in potentially higher costs in fixing them). The logical conclusion to above statement is that architectural views are synthesized in response to concerns and that they are analyzed to address (handle) those concerns. For architectural modeling this entails several challenges:

- need for support of a broad set of concerns,
- need for validation capabilities to ensure consistency between those architectural models, and
- need for an integrated toolset to support architectural modeling via multiple views.

The software architecture community has proposed a wide range of architectural description languages (ADLs) [10] over the past years and has therefore delivered a range of mechanisms to model and evaluate development concerns. However, these ADLs commonly assume a single (or primary) modeling language, making their integration rather difficult (e.g., ACME [7] had little success in providing a common architectural meta language). Having independent models, however, implies the possibility of creating inconsistencies due to the lack of information sharing. Assumptions and definitions common to multiple views must be defined and maintained consistently. In other words, having inconsistent assumptions about a

system's expected environment voids the correctness and usefulness of views and thus also renders invalid all solutions based on those views. For that purpose, architectural models need to be integrated with themselves and adjacent development models (e.g., requirements, design, etc.) to ensure overall development consistency. To this end, the IEEE P1471 standard confirms that an "[architecture description] should contain an analysis of consistency across all of its architectural views." Although standards such as P1471 have identified the need for consistency checking between architectural descriptions, they do not specify methods required to do that.

2 CLASSIFICATION

To address view integration, we need to consider the classification of views first. Basically we distinguish between *Fixed Views* and *View-Independent Models*: Fixed views are stand-alone and support their interaction with other views only through some explicitly defined form of data and/or control integration. Most (if not all) architectural definition languages fall into the category of fixed views. View-independent models, on the other hand, incorporate all relevant information about multiple views under one common roof with the advantage that information must not be exchanged or updated explicitly.

The trade-off between them is expressivity and consistency: If one accepts the idea that the drivers of architecting are stakeholders and their concerns, then view-independent models make the assumption that all stakeholders' concerns can be uniformly captured in a single representational scheme. If they can, then having a single model greatly simplifies the "integration problem". Although comprehensive models have emerged in the recent past, these models do not qualify as being view-independent. Take for instance UML (the Unified Modeling Language [11] [9]), which incorporates a number of views such as class diagrams, state diagrams, and sequence diagrams. Even though UML incorporates these views under one common meta-model, the actual storage does not exclude redundancy. It does not violate the UML notation to create two classes with a particular relationship between them and have their instances (objects) contain contradictory relationships. What the UML meta-model has achieved is not view integration but only view representation under a common roof.

On the other hand, requiring a view-independent representation of having no (or only minimal) model redundancy is somewhat of an utopia, however a desirable one since it promises working with multiple views without having to deal with consistency issues. The classification from fixed views to view-independent models is not a discrete one but continuous and manifests itself through three major stances:

- Constructive stance: Development views of systems are individually constructed. To understand the model is to understand the sum of all its views and their relationships.
- Projective stance: Views are projected out of the model to allow its inspection and/or manipulation. The model itself is all-comprehensive and views are needed to extract the *essence* of particular concerns.
- Decorative stance: Development views may be constructive and projective. Here a base representation exists that is annotated with additional information supporting a limited form of view projection.

Whereas the constructive and projective stances represent the extremes of view integration (none or full), the decorative stance takes the middle ground. Our work has shown that a decorative integration approach can be used even if the models and views were originally designed for a constructive stance (e.g., as in case of most ADLs and UML).

3 DECORATIVE VIEW INTEGRATION

We have devised and applied a view integration framework, accompanied by a set of activities and techniques to identify mismatches between architectural and/or design views in a more automatable fashion [3] [4]. Our view integration framework approach exploits the 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 analyze such constraints and, thereby, analyze the consistency across views.

To enable tool integration, we chose the UML meta model as a foundation for our model repository. The repository contains all relevant product information about the designed software system. Having used UML as a meta model does, however, not exclude using ADLs in our integration approach. To this end, we have found ways to integrating ADLs (e.g., C2 [12]) in UML [5]. Note that we adopted the use of UML's extensibility mechanism to represent C2 ADL concepts foreign to UML [1]. Although this mechanism has limitations it is still sufficient for our purposes here.

To manipulate and analyze the model repository, we use Rational Rose as well as our analysis tool called UML/Analyzer [3]. Our framework makes extensive use of model transformation to simplify information exchange and comparison. For instance, in order to compare the two user-defined views A and B, we could either a) compare them directly; b) transform (convert) A into 'something like B' so that A becomes easier comparable to B; c) transform B into 'something like A' so that B becomes easier comparable to A; or d) transform both A and B into 'something like C' so that they are easier to compare in the context of C (see Figure 1). Whereas A and B correspond directly to the types of views we wish to compare, C introduces a new type and could represent some formal

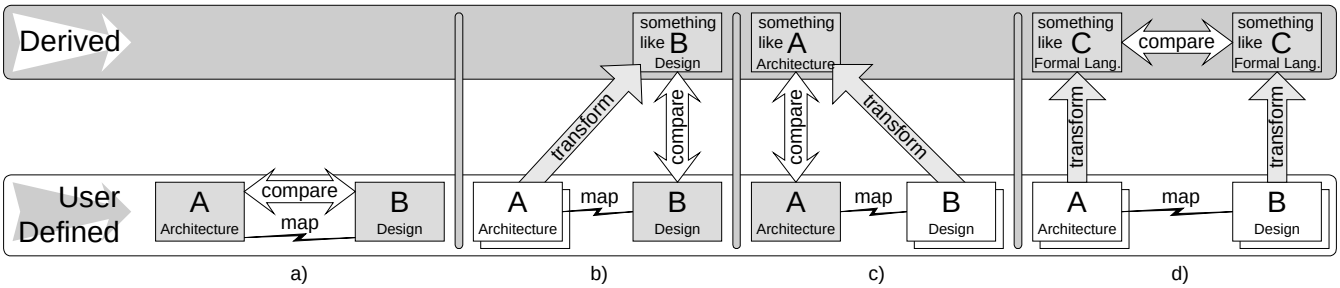

Figure 1. View Transformation and Mapping to Complement View Comparison (Differentiation)

language (e.g., [13] and [6]) or interchange format (e.g., [7]) with which A and B become easier comparable. In case of our C2 to UML integration we followed the third scenario (c) in Figure 1 of transforming the design into an architecture to enable a direct comparison between them.

On first glance, it may appear that we have adopted a constructive stance in our view integration approach which is not surprising since the models we used were of that stance. However, this was not our original goal since we wanted to avoid model redundancy altogether. Although it is theoretically possible to make UML view-independent we found that this would require a complete re-construction of its meta model; something we were not willing to do since that would violate the UML standard. Restricting ourselves to stay within the UML standard required that we used UML views under the constructive stance as they were created. We therefore made extensive use of transformation to enable multiple views to communicate and share information. A pure constructive stance in integrating UML models does, however, not come without a price tag. Initially, we defined and adopted simple transformation methods such as class abstractions, sequence to state consolidation [4], or C2 to object translation [1]. These simple transformation methods proved to be adequate in handling information exchange. However, we found that using those simple transformation techniques resulted in an extremely severe scalability problem caused by transformation redundancy as the following example illustrates:

Assume that we are constructing a software system and at its current state we have created say 100 user-defined views. Using transformation, we can easily come up with thousands derived views using various view combinations and transformation paths (e.g., note that transformation in theory can take any two views and transform them to another *derived* third view; those derived views in turn could be further derived creating a theoretically infinite number of transformation combination). Assume now that we modify a single model element in one of the 100 user views. The consequence would be quite disturbing: not only would we have to make sure this change is properly propagated to the other 99 user-defined views (for consistency), we now would also have to update *all* the derived views that we had computed from these user-defined views since they may have become inconsistent as well. In other words, view transformation on constructive models causes an enormous diseconomy of scale in that the number of views that are needed to be maintained consistently increases non-linearly over time. Under those circumstances it may seem impractical to use *any* automated view integration method that relies on transformation (or constructive views for that matter).

One may, however, suggest a straightforward way of avoiding the above problem by creating derived views on a need basis and discarding them thereafter. This approach would, however, create a scalability problem of another kind since the transformation overhead would increase through the lack of reuse. Thus, we are faced with the conflicting issues of wanting to preserve derived information without having to worry about keeping them consistent. This challenge can be addressed through a decorative stance in dealing with multiple views.

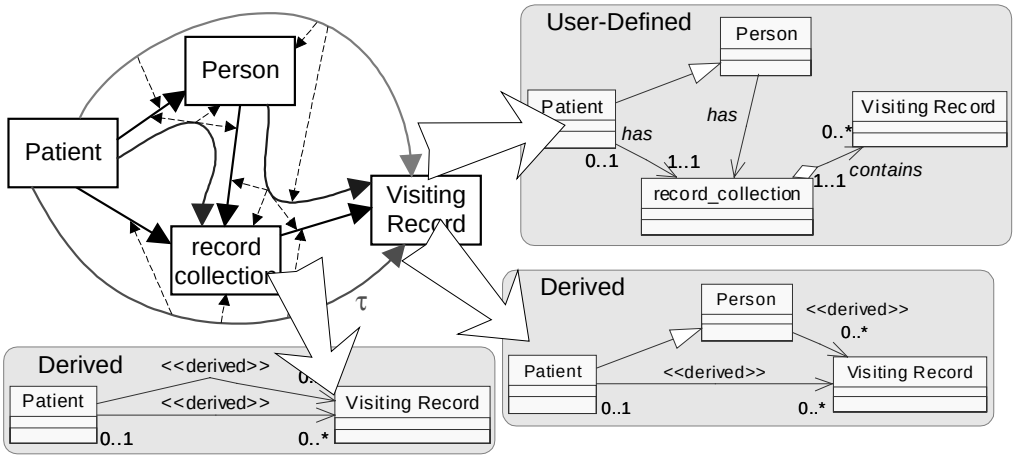

Figure 2. View Redundancy and its View-Independent Representation [3]

Applying the decorative stance, we create little pockets of view-independent models; or to be precise *derived-view-independent models*. Instead of creating separate spaces for derived views, their information is seamlessly integrated into the user-defined views. Derived views are then treated as *perspectives* and computing perspectives is computationally not very expensive. This approach is depicted in Figure 2. There the upper-right corner depicts a user-defined concrete view in the form of an UML class diagram. The two derived views in the lower half were generated automatically via the UML/Analyzer tool and represent abstractions of the user-defined view. Instead of storing derived information separately from user-defined information, we integrated them into a (more) *view-independent representation* as depicted on the upper-left. The image there shows a storage alternative for the three user-defined/derived views which has indeed view-independent characteristics since, for instance, we could change the name of the class *record collection* to *records* without requiring any additional consistency work in updating the derived views. However, note that this example does not create a fully view-independent model. If the class *record collection* is deleted then the abstracted relationship τ in the derived view still becomes inconsistent.

4 CONCLUSION

This paper discussed the need for architectural integration to support multiple stakeholders/concerns during software evolution. Views are an excellent mechanism for capturing stakeholder invariants. Further, the separated nature of views makes it much easier to manage them independently. A side effect of multiple views is the need of integrating them; foremost in order to maintain their consistency. We proposed a decorative approach in dealing with view integration since this approach can be made to work with regular constructive views but it minimizes the consistency-checking overhead and this improves scalability (the example used in this paper shows such a case). In [4] we describe our view integration framework in detail.

ACKNOWLEDGEMENTS

We wish to thank Barry Boehm and Nenad Medvidovic for helpful comments.

This research is sponsored by DARPA through Rome Laboratory under contract F30602-94-C-0195 and by the Affiliates of the USC Center for Software Engineering: http://sunset.usc.edu/CSE/Affiliates.html.

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. Boehm, B. W., Bose, P., Horowitz, E., and Lee, M. J.: "Software Requirements As Negotiated Win Conditions," *Proceedings of the International Conference on Requirements Engineering*, April 1994, pp.74-83.
3. Egyed, A.: "Using Model Transformation to Detect Inconsistencies between Heterogeneous Views," *submitted to the 8th Conference on Foundations of Software Engineering (FSE 8)*, 2000.
4. Egyed, A.: "Heterogeneous View Integration and its Automation," PhD Dissertation, University of Southern California, Los Angeles, CA, May 2000.
5. Egyed, A. and Medvidovic, N.: "A Formal Approach to Heterogeneous Software Modeling," *Proceedings of 3rd Foundational Aspects of Software Engineering (FASE)*, March 2000.
6. Finkelstein A., Kramer J., Nusibeh B., Finkelstein L., and Goedicke M.: Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal on Software Engineering and Knowledge Engineering*, 1991, 31-58.
7. Garlan, D., Monroe, R., and Wile, D.: "ACME: An Architecture Description Interchange Language," *Proceedings of CASCON'97*, November 1997.
8. IEEE Architecture Working Group: "Recommended Practice for Architectural Description," *IEEE P1471/D5.2 Information Technology Draft*, December 1999.
9. OMG: Unified Modeling Language Specification Version 1.3. OMG, 1999.
10. Perry D. E. and Wolf A. L.: Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 1992.
11. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.
12. 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.
13. Wang, E. Y. and Cheng, B. H. C.: "A Rigorous Object-Oriented Design Process," *Proceedings of the International Conference on Software Processes (ICSP5)*, June 1998.

For questions and comments, please send mail to:
Alexander Egyed at aegyed@sunset.usc.edu.