Published in Proceedings of 1st Workshop on Evaluating Software Architecture Solutions (WESAS), Irvine, CA, May 2000.

Validating Consistency between Architecture and Design Descriptions

Alexander Egyed
Computer Science Department
University of Southern California
Los Angeles, CA 90007, USA
aegyed@sunset.usc.edu

Special-purpose languages like ADLs [1] are very useful for modeling difficult and elusive concerns. ADLs provide a wide range of powerful and rigorous methods and frequently also support the analysis and simulation of problems or solutions. For general-purpose modeling, using ADLs may sometimes be an overkill. We therefore believe that general-purpose models like the Unified Modeling Language (UML) [2] are useful complements to ADLs. UML was defined in such a manner that it can be understood by a large population of software developers and beyond. The unified nature and simplicity have made UML the leading object-oriented design language there is today. Further, UML design constructs map more easily to the actual source code of a system. To that end, UML provides a variety of views, such as class, object, sequence, activity, statechart, and use-case views. Those views span a wide range of concerns and support structural, behavioral and scenario modeling.

In [3], we have investigated the issue of architecture to design integration in the context of the C2 ADL [4] and the Unified Modeling Language (UML). There, we discussed how to transform C2 architecture descriptions into (high-level) UML designs. Since designs are refinements of architectures and those designs are likely further refined into lower-level designs (and implementations), they could become inconsistent over time. This is particularly likely if refinements are done manually.

Figure 1 depicts a simple scenario of a part of a cargo routing application. The top-most row shows two architectural components of that cargo router, called *Port* and *Warehouse*.

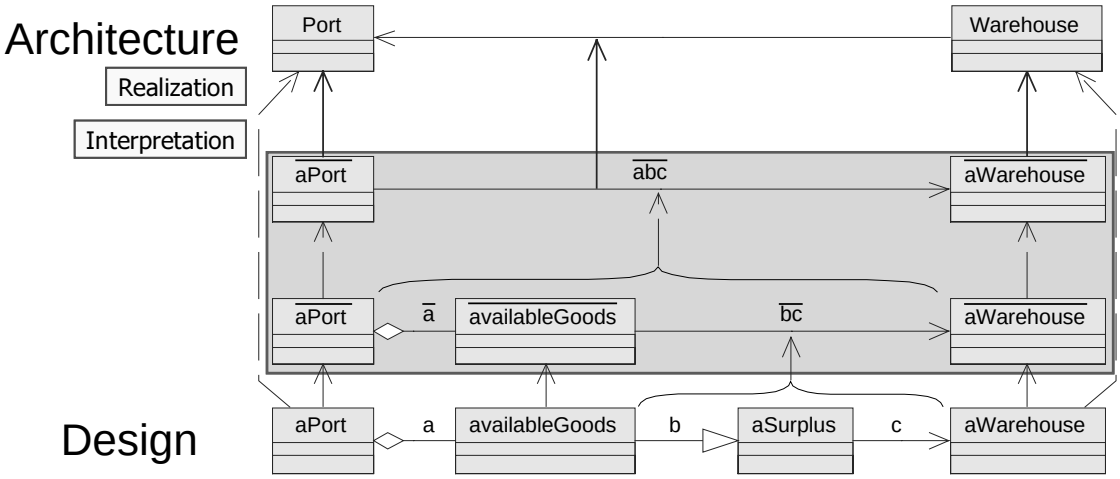

Figure 1. Consistency Checking between Architecture and Design Elements

The link between *Port* and *Warehouse* indicates that *Warehouse* accesses services provided by *Port*. The bottom-most row shows a design that is supposed to implement above architectural specifications. The design describes the relationships between the four design elements *aPort*, *availableGoods*, *aSurplus*, and *aWarehouse* using UML class diagram concepts.

In our work on view integration [5], we have shown techniques on how to ensure the consistency between diagrams. As such, we have investigated ways of describing and identifying the causes of modeling inconsistencies across UML and architectural views and have created a view integration framework, accompanied by a set of activities and techniques, for identifying inconsistencies in an automatable fashion. Our framework makes use of intermediate models into which diagrams are converted. Those intermediate models are usually generated via transformation methods. Figure 1 (middle rows) shows two intermediate models that were automatically generated by abstracting the design (bottom) via our UML/Analyzer tool. The details of our transformation method are not relevant here and can be found in [6]. What is relevant here is how the transformation results (derived model elements) relate to the user-defined model elements of the design and architecture. In particular we are interested in the realization/interpretation traces that were generated between the highest abstraction of the design and the architecture. Based on those traces, we can find inconsistencies.

Figure 2 depicts two (in)consistency rules. The rules describe conditions, if violated, indicate inconsistencies. For instance, the first rule states that if a relation is an interpretation (derived via transformation) and that relation has been generated via an abstraction method (thus is an abstraction) then this relation must have a realization. In Figure 1, we have only one relation that is both an interpretation and an abstraction, and that relation is "abc." The relation "abc" is an interpretation of the design and it was abstracted from the design. Since that relation also has a realization (the relation from *Warehouse* to *Port* on the architecture level), there is no inconsistency of this type in the figure.

The second rule states that if there is a realization that is also an interpretation, an abstraction, and has a realization then the realization of the destination of that relation must be same as the destination of the realization of that relation. In Figure 1, the relation "abc" is an interpretation, an abstraction, and has a realization. However, the destination of "abc" (*aWarehouse*) and its realization (*Warehouse*) is not equal to the realization of "abc" (arrow on architecture level) and its destination (*Port*). The violation of that rule indicates an inconsistency in that the direction of the concrete relation does not match the direction of the abstract relation. Since the concrete relation is part of the design and the abstract relation is part of the architecture, an inconsistency between the architecture and design is found.

Concrete relation has no corresponding abstraction:

$\forall$ r $\in$ relations, is_interpretation(r) $\wedge$ is_abstraction(r) $\Rightarrow$ realization(r) $\neq$ NULL

Direction of concrete relation does not match abstraction:

$\forall$ r $\in$ relations, is_interpretation(r) $\wedge$ is_abstraction(r) $\wedge$ realization(r) $\neq$ NULL $\Rightarrow$ realization(destination(r)) = destination(realization(r))

Figure 2. Consistency Rules between Abstraction and Refinement

Above example showed how transformation and analysis can help in identifying inconsistencies among abstractions and refinements. Without consistent refinement, the effort spend in creating, analyzing, and simulating architectural specification would be wasted since only consistent refinement can guarantee that proven architectural properties (such as security, reliability, and performance) have actually been implemented correctly. In our work on view analysis, we have described 20 types of inconsistencies that can happen between an abstract model and a concrete model. Using our abstraction methods as well as our consistency rules, inconsistencies between architecture and design can be identified in a more automated fashion.

Acknowledgements

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

References

1. Medvidovic, N., and Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, to appear, 2000.
2. G. Booch, I. Jacobson, and J. Rumbaugh, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.
3. Egyed, A. and Medvidovic, N., "A Formal Approach to Heterogeneous Software Modeling", Proceedings of Foundational Aspects of Software Engineering (FASE), Berlin, Germany, 2000.
4. 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, vol. 22, no. 6, pp. 390-406, 1996.
5. Egyed, A. "Heterogeneous View Integration and its Automation," to appear, Ph.D. Dissertation, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781, USA, 1998
6. Egyed, A. and Kruchten, P. "Rose/Architect: a tool to visualize software architecture," Proceedings of the 32nd Annual Hawaii Conference on Systems Sciences, 1999.