Published in the Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE), San Diego, USA, November 2001, pp. forthcoming.

Scalable Consistency Checking between Diagrams – The VIEWINTEGRA Approach

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 231
Marina Del Rey, CA 90292, USA
aegyed@ieee.org

Abstract

The Unified Modeling Language (UML) supports a wide range of diagrams for modeling software development concerns. UML diagrams are independent but connected; their meta-model describes them under a common roof. Despite the advances of UML, we found that the problem of ensuring consistency between UML diagrams has not been solved. In the past years, we have developed an approach for automated consistency checking, called VIEWINTEGRA. Our approach provides excellent support for active (preventive) and passive (detective) consistency checking. We make use of consistent transformation to translate diagrams into interpretations and we use consistency comparison to compare those interpretations to other diagrams. Our approach was applied to a number of applications where we found the separation of transformation and comparison to be highly beneficial in addressing consistency-checking scalability & usability issues. This paper introduces our UML-based transformation framework, discusses how it aids comparison, and demonstrates how it improves consistency checking.

1. Introduction

Models and diagrams are useful in separating concerns and handling complexity by providing different viewpoints. Diagrams have in common that they break up software development into smaller, more comprehensible pieces utilizing a divide and conquer strategy. These pieces are related in the system they describe and, combined, they form a model description of a system [2].

The major drawback of "diagram-centric" software engineering is that development concerns cannot be truly investigated individually since they tend to affect one another. If a set of issues about a software system is investigated, each through its own set of diagrams then their combined correctness requires that common assumptions and definitions within those diagrams are recognized and maintained in a consistent fashion. Consistency is complicated by the very advantage of diagrams – their separation. Although the separation of diagrams is beneficial in supporting separate concerns, it is also bad because it induces duplication (redundancy) of model information. Explicit mechanisms are required to ensure the consistency of all redundant information across all diagrams; but conceptual differences in meaning and language are barrier to consistency checking and complicate matters.

This work introduces a transformation-based approach to consistency checking. We discuss our approach in context of various UML diagram types (i.e., class, sequence, collaboration, object, statechart diagrams) at different levels of abstraction (i.e., high-level versus low-level diagrams). Our approach, called VIEWINTEGRA, is primarily aimed at detecting inconsistencies after they have been introduced. We take the stance that software diagrams are inherently ambiguous and inconsistencies often cannot be avoided [2,4].

In this paper, we will emphasize the important nature of transformation as a means of improving consistency-checking scalability. It is our finding that consistency checking between "n" diagrams does not require "n*n" number of transformations and/or comparisons but significantly fewer. It is also our finding that the separation of transformation and comparison makes comparison rules (consistency checking rules) simpler and less in numbers. Furthermore, separate transformation methods can also be used independently of consistency checking for reverse engineering and other needs.

2. The VIEWINTEGRA Approach

Diagrams are self-contained in describing individual aspects of software systems. No diagram is replaceable by any other diagram but diagrams are not independent of one another either. For instance, a sequence diagram describes interactions among objects whose types are classes. A sequence diagram thus is not independent from a class diagram. Similarly, a class diagram describes interactions that must also be captured in refinements. Abstract and refined diagrams are thus not independent from one another. In fact there are mutual dependencies between most diagrams implying that there is substantial

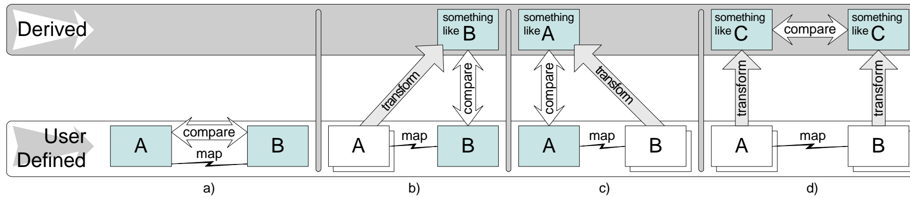

Figure 1. Model Transformation, Mapping, and Comparison

information redundancy. Unfortunately, redundancies imply the possibility of inconsistencies.

In an ideal world, diagrams are tightly integrated to avoid inconsistencies. In reality such tight integration is infeasible and often impractical [2]. This is especially then the case when redundancies are obscure and do not follow simple one-to-one mappings. There are also non-technical reasons in favor of allowing inconsistencies to happen. For instance, there are situations where inconsistencies are evolutionary necessities in that development uncertainties are often purposely left ambiguous to not over-constrain the design [4].

Our consistency-checking approach separates consistent transformation from consistency comparison. Consistent transformation ensures consistency via well-defined transformation steps where source diagrams are transformed into target diagrams in a manner that guarantees consistency. Consistency comparison, on the other hand, detects inconsistencies via well-defined comparison steps where source diagrams are compared to target diagrams to identify them. Consistent transformation enables "continuity" by bridging modeling information across the model barrier but it comes at the expense of evolutionary freedom in that re-transformation may potentially overwrite modifications. The alternative, consistency comparison, allows inconsistencies to be detected after they have been introduced and has the advantage that models can evolve separately. Comparison comes at the expense of model continuity in that consistency detection is not equivalent to consistency resolution (fixing an inconsistency is a form of model continuity).

Our approach – VIEWINTEGRA – combines the active nature of consistent transformation with the passive nature of consistency comparison. We make use of consistent transformation to convert source diagrams into the diagram type of the target so that transformations of the source diagrams are conceptually close to target diagrams they need to be compared with. We then use consistency comparison to compare the transformation of the source diagram with the target diagram. It must be emphasized that we do not use "third-party diagrams" to validate the consistency of UML diagrams. If we would like to compare a sequence diagram with a class diagram then we would either transform the sequence diagram into an "interpreted" class diagram followed by comparing the interpreted class diagram with the existing one (Figure 1 (b)); or we would transform the class diagram into an interpreted sequence diagram followed by comparing the sequence diagrams (Figure 1 (c)).

Our approach "inherits" increased model continuity from consistent transformation by bridging model information across the model barrier; and our approach "inherits" evolutionary consistency from consistency comparison by allowing inconsistencies to exist in models. If no single transformation method can convert a source diagram conceptually close to a target diagram then transformation methods may also be applied in series or in parallel (Figure 1 (d)). If no set of transformation methods yields comparable diagrams and no direct comparison is supported (Figure 1 (a)) then automated consistency checking is not possible.

3. Heterogeneous Transformation

For consistency checking we need transformation methods to bridge model information between all diagram pairs. Fortunately, it is not important in what direction to transform; it is not significant whether "A" is transformed to "B" or vice versa since A=B is equivalent to B=A. Despite this, a brute force approach to heterogeneous transformation (meaning the transformation between all diagram pairs) does not scale. Figure 2 shows six diagram types like statechart models, class models, or object models. Since those diagrams can be used to describe systems at different levels of abstraction, we have 11 groups of diagrams at our disposal (or more if additional levels of abstraction are considered). To ensure consistency, we thus require 55 transformation methods ("n*(n-1)/2" where "n" stands for the number of diagrams). We understand a transformation method to be a technique that converts one diagram type to another diagram type (i.e., convert sequence diagrams into class diagrams). A "brute force" method to consistency checking thus causes a tremendous overhead. Note that this non-scalability applies equally to comparison since 55 comparisons methods are needed to perform consistency checking on the 11 given models.

On first glance, diagram transformation appears dependent on the types of diagrams transformed. For instance, abstracting a class diagram may seem different from abstracting a statechart diagram. Although this observation is generally true, we did find that diagrams can be grouped into categories and transforming between those categories often involves similar techniques. Figure 2 depicts the three major dimensions we identified, going from specific to generic (e.g., object to class), from behavior to structure (e.g., sequence to object), and from low-level to high-level (e.g., low-level class to high-level

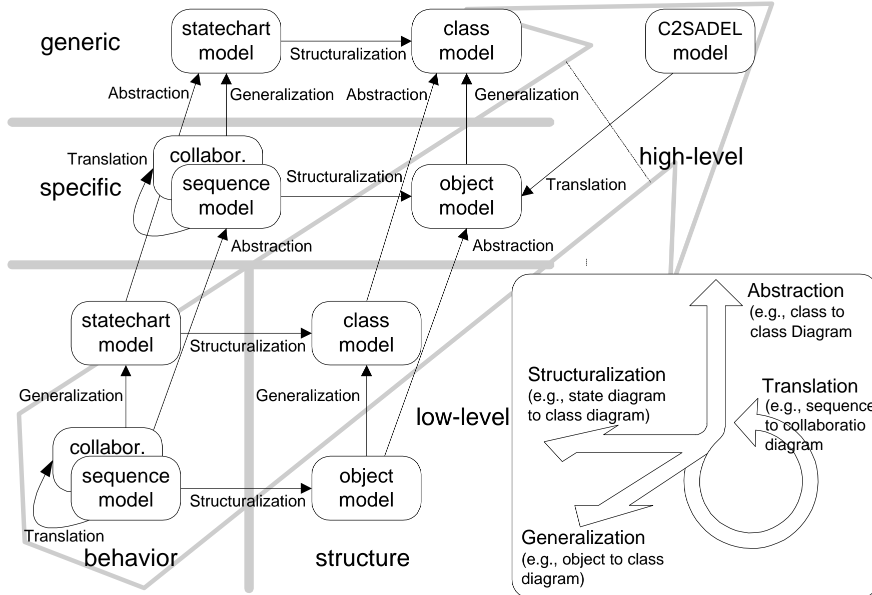

Figure 2. Transformation Framework for Scalable Comparison

class). All types of diagrams (and models) we analyzed (including some *outside* the UML domain such as C2SADEL [5]) can be positioned relative to these three dimensions.

The first of the three dimensions in Figure 2 is the low-level/high-level dimension. This dimension denotes the level of abstraction of diagrams. For instance an architectural model is usually more abstract (high-level) than a design model. The generic-specific dimension denotes universal validity of information. For instance, class diagrams describe generic relationships whereas sequence diagrams describe examples. The behavior-structure dimension denotes the extent in which behavior is dictated by structure. For instance, a sequence diagram describes interactions in a step-by-step manner including their ordering whereas a class diagram only indicates the existence of interactions but does not describe when they happen. Having three dimensions of diagrams implies four types of transformation axes: *Abstraction* to capture conversion from low-level to high-level, *Structuralization* to capture conversion from behavior to structure, *Generalization* to capture conversion from specific to generic, and, *Translation* to capture other forms of conversions (i.e., as in the translation of sequence to collaboration diagrams).

These dimensions denote “natural boundaries” between diagrams. Our observation about the existence of those boundaries is not surprising since others have made similar distinctions in the past [1,3,4]. However, we took these boundaries and dimensions a step further and build a transformation and comparison framework around them. Our approach only requires 10 transformation methods, a small subset of the 55 potential transformation methods between all diagrams depicted in Figure 2. Nonetheless, our 10 transformation methods still support the comparison between all diagram pairs and it facilitates reuse which improves scalability.

4. Semantic Implications

Most obviously, our framework reduces the entry barrier to comprehensive, automated consistency checking by only requiring the automation of 10 transformation methods as opposed to 55. Our approach also does not create the overhead of using third-party, non-standard languages; although it is possible to do so if needed. And, our approach is useful beyond consistency checking since transformation methods can also be used for other automation purposes (i.e., forward and reverse engineering).

4.1 Transformations for Comprehensive Comparison

Thus far, we only discussed 10 transformation methods but we left it open how to do pair-wise consistency checking between all diagram types which normally require 55 transformation methods. To enable these additional transformations, we use serial transformation. For instance, if we wish to compare a sequence diagram with the a class diagram then we need to structuralize the sequence diagram into an object diagram followed by generalizing that object diagram into a class diagram (serial); and we need to generalize the sequence diagram into a statechart diagram followed by structuralizing that statechart diagram.

Ideally, both transformation paths should yield the same results and thus one transformation path (i.e., sequence to object to class diagram) should be sufficient to compare sequence diagrams with class diagrams. In practice, we find that object diagrams and statechart diagrams individually are not ideal “intermediate representations” to bridge the gap between sequence and class diagrams. Thus, there is a benefit in following both transformation paths. For consistency checking this implies that two interpretations are generated if a sequence diagrams is compared to a class diagrams. For comparison this implies that both interpretations have to be compared to the class diagram.

Serial transformation not only increases the effectiveness of our 10 transformation methods but also has the benefit of reuse. Recall that the example of sequence to class transformation generated intermediate statechart and object diagrams. These intermediate diagrams are needed for subsequent transformation(s) so that the sequence dia-

gram becomes comparable to the class diagram; however, those intermediate diagrams can also be compared directly to other existing statechart and object diagrams if such should exist. Generating intermediate diagrams is therefore not a computational overhead but instead very useful for subsequent comparisons with other diagrams. This form of reuse would not be possible if all 55 required transformation methods were implemented independently.

Despite the benefits, serial transformation does not enable the transformation of all diagram pairs. Given the uni-directional nature of most of our transformation methods, it is not possible to transform an object diagram into a statechart diagram or vice versa. To check for consistency between them, we use parallel transformation (Figure 1 (d)). Parallel transformation looks for a common denominator for comparison. To compare object and statechart diagrams, we transform both of them into class diagrams and compare the resulting two class diagrams for consistency. Like with serial transformation, parallel transformation can use and produce reusable intermediate diagrams for other transformations and/or comparisons.

4.2 Simplicity of Consistency Rules

Comparison takes diagrams and compares them to identify differences. Comparison is supported by rules (inconsistency rules) that describe if and how much diagrams may differ. Violations of those rules indicate inconsistencies. Consistency rules supporting our framework are very simple since our framework always guarantees that diagrams are compared within a single diagram type. For instance, comparing a sequence diagram with a class diagram results in comparing class diagrams only – the original one and the transformation. This feature has another desirable side effect: instead having to define inconsistency rules for class/sequence diagrams *and* for class/collaboration diagrams separately, VIEWINTEGRA uses the exact same rules for both because ultimately both sequence and collaboration diagrams are interpreted as class diagrams for comparison.

5. Conclusion

This paper presented the VIEWINTEGRA approach to consistency checking among UML diagrams. We presented a transformation framework and demonstrated its use in context of five UML diagram types (class, object, sequence, collaboration, and statechart diagrams). This paper particularly emphasized transformation-based consistency checking without the need of third-party, intermediate languages.

Our approach uses transformation to bring models closer to one another in order to simply comparison. Instead of implementing 55 transformation methods for comparing 11 UML diagram types, our approach only needs a subset containing 10 transformation methods. This not only reduces the entry barrier for automated consistency checking but it also enables reuse of interpretations during serial and parallel transformation. Separating transformation from comparison also makes consistency-checking rules simpler and less in numbers. Instead of defining 55 sets of consistency rules between 11 diagram types, our approach only defines 10 sets of consistency rules. It is therefore our observation that our approach improves the scalability of consistency checking without sacrificing usability. Our transformation framework can also be used for forward and reverse engineering.

Currently five of the ten introduced transformation methods are automated and tool supported. Future work is to provide full automation and to integrate other types of diagrams.

Acknowledgements

We wish to thank Dave Wile, Nenad Medvidovic, Barry Boehm, Bashar Nuseibeh, and all anonymous reviewers for insightful comments and discussions. This work was supported by DARPA under agreements F30602-00-C-0218, F30602-99-1-0524, and F30602-00-C-0200.

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),* Fort Collins, CO, Oct. 1999.

[2] Balzer, R., "Tolerating Inconsistency," *Proceedings of 13th International Conference on Software Engineering (ICSE-13),* pp. 158-165, May 1991.

[3] Moriconi, M., Qian, X., and Riemenschneider, R. A., Correct Architecture Refinement *IEEE Transactions on Software Engineering*, vol. 21, pp. 356-372, Apr, 1995.

[4] Nuseibeh, B., A Multi-Perspective Framework for Method Integration 1994. Imperial College of Science, Technology and Medicine.

[5] 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, pp. 390-406, 1996.