Proceedings of the 2nd International Workshop on Living with Inconsistencies (IWLWI), co-located with ICSE 2001, Toronto, Canada, May 2001.

Transformation, Ambiguity, and Trivialization

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 231
Marina Del Rey, CA 90292, USA
+1 310 578 5350

aegyed@acm.org

1. INTRODUCTION

We have developed a consistency checking approach that 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 where source models are compared to target models to identify inconsistencies. By combining consistent transformation and consistency comparison we found that their respective disadvantages can be leveraged. Whereas consistent transformation enables automated continuity of modeling information (thus avoids error-prone, manual, and repetitive activities in re-capturing model information) it nevertheless comes at the expense of evolutionary freedom in that either source and target models cannot be modified after transformation since re-transformation may potentially overwrite those modifications. The alternative, consistency comparison, allows inconsistencies to be detected "after the fact" (thus also avoids error-prone, manual, and repetitive activities in comparing models) and has the advantage that models can be evolved separately with the potential of exposing them to inconsistencies (a form of living with inconsistencies) due to their separate evolution. Consistency comparison however comes at the expense of model continuity in that inconsistency detection is not equivalent to inconsistency resolution.

We have developed a hybrid consistency checking approach that combines the active nature of consistent transformation with the passive nature of consistency comparison; complementing model continuity with evolutionary consistency. We refer to our transformation-based consistency checking approach as *IViTA*. (Inter-VIew Transformation and Analysis). Note that our approach is conceptually similar to consistency checking approaches like VisualSpecs [1] or JViews [6], 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 one-to-one manner (i.e., between low-level and high-level class).

Our approach does not invent new intermediate languages (i.e., as in MViews) and our approach does not use transformation to convert model elements into a "comparison language" (i.e., VisualSpecs) but instead we 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 a class diagram with a corresponding source code, our approach would either reverse engineer the source code to yield an "interpreted" class diagram followed by comparing the interpreted class diagram with the existing one; or our approach would generate a source code interpretation out of the class diagram followed by a comparison of both versions of the source code. Thus there is no need for new model languages although there are cases where we can envision advantages in having them. It is thus not our emphasis to bring models closer to comparison languages but instead to bring models closer to one another. And there is increasing support for that: Koskimies et al. [8] created a sequence to statechart transformation methods that use groups of sequence diagrams (like test cases showing order of method calls among multiple objects) to generate statechart diagrams (depicting life cycles of software components); Ehrig et al. [5] came up with another model transformation method that consolidates collections of object diagrams to reason about their differences. They then map those differences to method calls (as described in class diagrams) to reason about the impact those methods have onto objects; and Egyed-Kruchten [4] developed an abstraction method on how to eliminate low-level classes to yield high-level abstractions.

Our approach uses those kinds of transformation methods to convert model elements into intermediate models (called interpretations) in such a manner that they (or pieces of them) can be compared to other model(s). Generally, transformation allows conceptually different models to be compared directly. Thus, we aim at creating an environment where two originally different types of models can then be compared in context of one of their types (i.e., convert the sequence diagrams to statechart diagrams to simplify its comparison with other statechart diagrams). Our comparison language is thus dependent on the types of models we wish to compare.

2. TRIVIALIZATION AVOIDANCE

Although transformation-based consistency checking can combine the advantages of consistent transformation and consistency comparison, it cannot necessarily eliminate all their flaws. In the past three years, we have inspected a large number of third-party software models and we have observed that there are numerous situations where modeling is impaired by ambiguities. We found that ambiguities can be encountered during consistent transformation and consistency comparison because of missing trace dependencies and indeterminant model transformations. In [3], we argue that in both cases ambiguities manifest themselves in complex many-to-many mappings which need to be dealt with during consistency checking.

In investigating the issue of trivialization in context of *IViTA*, we found that the foundation for trivialization is already laid during

transformation. Hunter and Nuseibeh [7] argue that trivialization is the result of an inconsistency during comparison introducing a negation that potentially makes everything true ($A \wedge \neg A$). During transformation, however, we found that indeterminant situations may result in ambiguities that may lead to an increased likelihood of trivializations. We encountered this case when a transformation method is indecisive as to what the correct transformation result for a given source model is. For instance, if during transformation it is found that a model element might be either "A" or "B" ($A \vee B$) then during comparison (consistency checking) we might encounter the situation $\neg A \wedge (A \vee B)$. Although the ambiguity ($A \vee B$) has a 50% change of being B, the expression $\neg A \wedge (A \vee B)$ will always result in trivialization.

It is our belief that ambiguity increases the likelihood of trivialization –ambiguity thus amplifies trivialization. We argue that in minimizing ambiguity problems we also minimize trivialization problems. In [3], we present rules for ambiguous reasoning that are tied to a maximum bi-partite matching algorithm that guarantees minimal ambiguity.

3. TRIVIALIZATION CONTAINMENT

Although, we found that every step should be taken to reduce the trivialization problem much like every step should be taken to reduce the ambiguity problem, minimizing ambiguities is not equal to eliminating ambiguities. Thus once trivialization is encountered, we are faced with the problem on how to deal with it. Naturally, we could employ techniques like quasi-classical logic [7] but, in context of *IViTA*, we also found our transformation framework to enable the containment of trivialization.

In our investigation of UML diagrams, we have identified three major transformational axes (see Figure 1). Views can be seen as *high-level* or low-level, *generic* or *specific*, and *behavioral* or *structural* [2]. The high-level/low-level dimension denoted differences between model elements of different levels of abstraction. For instance, low-level class diagrams realize high-level class diagrams through refinement. The generic-specific dimension denotes the generality of modeling information. For instance, a class diagram naturally describes a relationship between classes that must always hold, whereas an object diagram describes a specific scenario (a subset). Finally, the behavior-structure dimension takes information about a system's behavior to infer its structure. For instance, test scenarios (which are behavioral) depict interactions between objects (structural) and may thus be used to infer structure.

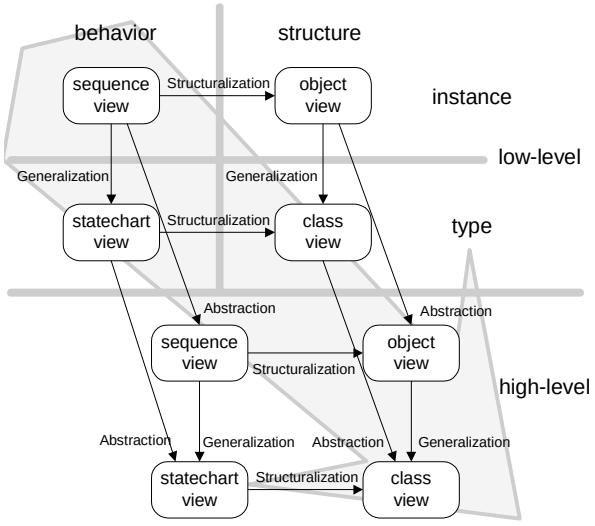

Figure 1. Transformation Framework to Contain Trivialization

We use the transformation infrastructure to optimize transformations in *IViTA*. For instance, if we would like to compare a low-level sequence diagram with a high-level class diagram, we could structuralize the sequence diagram to an object diagram, abstract the (still low-level) object diagram to a high-level object diagram, and finally generalize from the object diagram to the class diagram. Our transformation framework takes advantage of similar transformation properties among the three defined transformation axes, however, more significantly for trivialization; our transformation framework restricts needed comparisons. For instance, if we compare a low-level object diagram with its low-level class diagram and that low-level class diagram with a high-level class diagram then there is little value in also comparing the low-level object diagram with the high-level class diagram (the likelihood of same of similar inconsistencies encountered is much greater). Thus, if the comparison of the object diagram with the low-level class diagrams induces a trivialization then it does not affect the subsequent comparison of the low-level class diagram with the high-level class diagram.

4. CONCLUSIONS

This paper presented a tool-supported, transformation-based consistency checking approach and discussed how it can be used minimize trivialization and how to contain it. Although the proposed solutions do not eliminate the trivialization problem, they nevertheless ease it. Future work is to complement our approach with a technique on how to resolve trivialization problems.

[1] Cheng, B. H. C., Wang, E. Y., and Bourdeau, R. H., "A Graphical Environment for Formally Developing Object-Oriented Software," *Proceedings of International Conference on Tools with AI*, Nov. 1994.

[2] Egyed, A., Heterogeneous View Integration and its Automation 2000. University of Southern California.

[3] Egyed, A., "Taming Ambiguity to Overcome the Model Consistency Barrier," *submitted to European Conference on Software Engineering and Foundations of Software Engineering (ESEC/FSE)*, Sept. 2001.

[4] Egyed, A. and Kruchten, P., "Rose/Architect: a tool to visualize architecture," *Proceedings of the 32nd Hawaii International Conference on System Sciences*, Jan. 1999.

[5] Ehrig, H., Heckel, R., Taentzer, G., and Engels, G., A Combined Reference Model- and View-Based Approach to System Specification *Journal of Software Engineering and Knowledge Engineering*, vol. 7, pp. 457-477, 1997.

[6] Grundy, J., Hosking, J., and Mugridge, R., Inconsistency Management for Multiple-View Software Development Environments *IEEE Trans. Soft. Eng.*, vol. 24, Nov, 1998.

[7] Hunter, A. and Nuseibeh, B., Managing Inconsistent Specifications: Reasoning, Analysis, and Action, *Trans. Soft.Eng. and Methodology*, vol. 7, pp. 335-367, Oct, 1998.

[8] Koskimies, K., Systä, T., Tuomi, J., and Männistö, T., Automated Support for Modelling OO Software, *IEEE Software*, vol. pp. 87-94, Jan, 1998.