Fine-Tuning Model Transformation: Change Propagation in Context of Consistency, Completeness, and Human Guidance

Alexander Egyed, Andreas Demuth, Achraf Ghabi, Roberto Lopez-Herrejon, Patrick Mäder, Alexander Nöhrer, and Alexander Reder

[1] Institute for Systems Engineering and Automation
Johannes Kepler University
Altenbergerstr. 69, 4040 Linz, Austria
{firstname.lastname}@jku.at

Abstract. An important role of model transformation is in exchanging modeling information among diverse modeling languages. However, while a model is typically constrained by other models, additional information is often necessary to transform said models entirely. This dilemma poses unique challenges for the model transformation community. To counter this problem we require a smart transformation assistant. Such an assistant should be able to combine information from diverse models, react incrementally to enable transformation as information becomes available, and accept human guidance – from direct queries to understanding the designer(s) intentions. Such an assistant should embrace variability to explicitly express and constrain uncertainties during transformation – for example, by transforming alternatives (if no unique transformation result is computable) and constraining these alternatives during subsequent modeling. We would want this smart assistant to optimize how it seeks guidance, perhaps by asking the most beneficial questions first while avoiding asking questions at inappropriate times. Finally, we would want to ensure that such an assistant produces correct transformation results despite the presence of inconsistencies. Inconsistencies are often tolerated yet we have to understand that their presence may inadvertently trigger erroneous transformations, thus requiring backtracking and/or sandboxing of transformation results. This paper explores these and other issues concerning model transformation and sketches challenges and opportunities.

Keywords: change propagation, transformation, consistency, variability, constraints, impact of a change

1 Introduction

There are many benefits to software and systems modeling but these benefits hinge on the fact that models must be internally consistent. However, for models to be consistent, changes (additions, removals, and modifications) must be propagated correctly and completely with reasonable effort. Unfortunately, a change is rarely a localized event. On the code level, changes tend to affect seemingly unrelated parts [1], considering the wider dimension of software engineering, changes affect everything from requirements, models, code, test scenarios, documentation, and more [2, 3]. Considering the even wider dimension of systems, changes in one discipline (and its models) affect other disciplines (and their models). Unfortunately, when it comes to change, designers simply lack the engineering principles to guide them. Model transformation provides the means for propagating knowledge from one model to another. It is intuitive to think of change propagation as a series of model transformations where either the changed model is re-transformed to other models or only the change itself is transformed. Yet, transformation methods have to overcome a range of challenges to support change propagation.

In this paper, we discuss the challenge of change propagation in software and systems models from the perspective of transformation. It should be noted that we do not believe that change propagation is fully automatable since creativity is a major part of this process – and being creative is what humans do best. But a human should enter a modeling fact once only. If knowledge is replicated across multiple models then this knowledge should be propagated automatically. If this knowledge is changed then it should be updated. Unfortunately, the diversity and inter-disciplinary nature of models rarely sees model elements to be replicated directly. Moreover, models typically do not (just) replicate knowledge from other models but also add their own, unique information. This implies that model transformation is rarely fully automatable. A simple analogy is the blueprint for a house. While the side view of a house cannot be derived from the front view (no automatic transformation), it is obvious that the height of the house must be the same in both views - a restriction that two modeling views impose on each other. This is intuitive because if models were derivable through other models then one modeling language could replace another (or one discipline with its models could replace another discipline with their models) – which is not desirable. The sole purpose of diverse modeling languages (with separate structure, behavior, scenarios, and more) is to depict modeling information from different points of views that may overlap in the knowledge they include but are meant to have unique parts also.

It is the objective of this paper to highlight challenges on how to automatically propagate changes across diverse, inter-disciplinary design models – an unsolved problem of vital interest to software and systems engineering disciplines. If during change propagation, the information needed is already present in the model (perhaps in a semantically different, distributed form) then a goal of change propagation is to transform that information (if possible) or to restrict possible changes in other models. If some information needed is not present in the model then the goal of change propagation is to elicit this missing information from the human designer in ways that do not obstruct/interrupt their work. The role of the designer is thus to instigate change propagation and guide it. The role of automation should be to reason about the logical implications of changes in context of diverse models.

2 Illustration and Problem

Figure 1 introduces a small example to illustrate change propagation in context of three UML diagrams. The class diagram (left) represents the structure of a movie player: a *Display* used for visualizing movies and receiving user input and a *Streamer* for decoding movies. The sequence diagram (right) describes the process of selecting a movie and playing it. A sequence diagram contains interactions among instances of classes (vertical life lines), here a user invoking *select* (a message) on the display lifeline of type *Display* which then invokes *connect* on the *streamer* lifeline of type *Streamer*. The movie starts playing once the *playOrStop* message occurs which is followed by *stream*. The state machine (middle) describes the allowed behavior of the *Streamer* class. It is depicted that after *connect*, the *Streamer* should toggle between the *stopped* and *playing* states. Change propagation can only be automated to the degree that failure to propagate changes is observable. Indeed, we believe that model constraints are the perfect foundation to understand failure to propagate [4] and change propagation should leverage from knowledge about constraints [5]. For example, imagine that the designer changes the model, say, by splitting the single *playOrStop* operation in the class diagram into two separate operations called *play* and *stop* (see top left of Figure 1). This change in the class diagram causes inconsistencies with the state machine and sequence diagram due to their continuing use of the old names. Such inconsistencies are the result of violations of constraints which govern the correctness within and among such diagrams (note that we use the terms *model* and *diagram* interchangeably). Table 1 depicts three such constraints, the first of which is described in more detail using the OCL constraint language [6]. It defines that the name of a message must match at least one operation in the class diagram – not just any operation but the one on the receiving end of the message (arrow head).

Constraint 1	**Name of message must match an operation in receiver's class** context Message inv: self.receiveEvent.covered->forAll(Lifeline l|l.represents.type.ownedOperation->exists(Operation o|o.name=self.name))
Constraint 2	**Name of statemachine action must match an operation in owner's class**
Constraint 3	**Sequence of messages must match allowed sequence of actions in state machine**

Table 1. Constraints are useful for Change Propagation

This paper suggests that knowledge about changes, combined with knowledge about possible transformations and constraints that govern the correctness of the models, results in a better understanding of change propagation. The example also makes it obvious that transformation must be possible partially or in form of alternatives if no unique answer is computable. Take for example the implication of the designer change onto the state machine. The two transitions *playOrStop* have to be changed – at least in name – however, it is not possible to automatically transform this change in the state machine because the designer has not provided enough information to infer this.

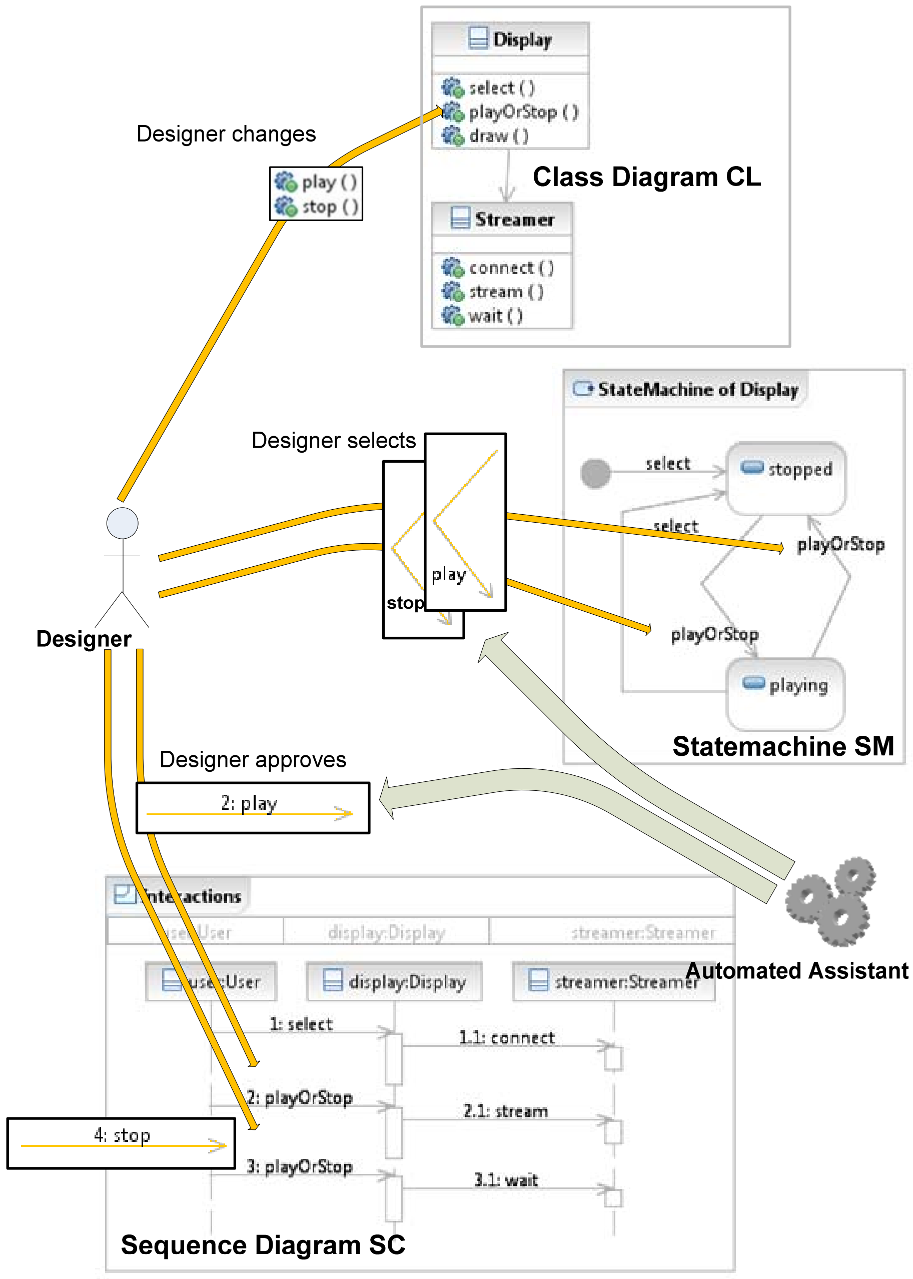

Figure 1. Engineer changes the class diagram and an Automated Assistant could suggest choices for how to change the state machine (partial automation). After the engineer selects one of these choices, the Assistant could change the sequence diagram by itself (complete automation).

Which *playOrStop* transition should be named *play*, which one *stop?* All we know is that with the *playOrStop* method gone, the same named transitions in the state machine are affected. It is our opinion that it is not useful to propagate “likely” changes and we advocate strongly against any approach that is not generic. As such, approaches that were to propagate changes based on a heuristic, such as minimizing the number of inconsistencies caused by change propagation, would be incorrect quite often. A trivial proof is the simple undo. The undo is likely the most effective way of eliminating an inconsistency (all we need to do is to undo the last change which apparently caused the inconsistency). While the undo would immediately “solve” the inconsistency, the undo would conflict with the designer’s intention most times. Of course, in case of the undo, this conflict between change propagation on one hand and quick inconsistency resolution is obvious. This issue, however, becomes trickier in context of changes that carry across multiple models. What we need is thus a generic mechanism to propagate precise and complete changes or, if not possible, to propagate precise and complete restrictions (=other kinds of constraints). That is, we may not know exactly how to change the state machine; however, we can reason precisely in what ways not to change the state machine. We could even compute a list of alternative, reasonable changes: for example, automated change propagation may suggest renaming the transitions in the state machine to either *play* or *stop* which the designer must then do manually by choosing between them. The designer then complements the inferable changes from the class diagram with missing information.

The same is true for the two messages in the sequence diagram in the right. Given the changes in the class diagram, we also cannot decide exactly how to rename the messages (if renaming is the designer’s way of resolving this inconsistency which is but one option). It is thus vital to combine knowledge from transformation (including alternatives and restrictions) across multiple models and knowledge of all inputs provided by the designer to understand his/her intentions. With every change made by the designer and with every intervention, the designer’s intention becomes increasingly better known. For example, if the designer selects one of the choices on how to rename the transitions in the state machine then, combined with the knowledge how the designer changed the class diagram, we can automatically infer what changes must happen to the sequence diagram. Concretely, if the designer selects the name *play* for the left *playOrStop* transition and *stop* for the right one then, based on the given constraints and the restrictions of both designer changes, we can automatically decide that the top *playOrStop* message in the sequence diagram must be renamed to *play* and the bottom one to *stop*. This conclusion would be the only one that would satisfy all constraints imposed in Table 1 because the state machine defines that the *play* transition must happen after either the *connect* or *stop* transition and the sequence diagram list the *connect* message before the *playOrStop* message.

Our goal should thus be an automated assistant with a well-defined methodology for reasoning about changes, their interpretations, and their combined propagation. The benefit of such an assistant was to request human intervention only when necessary to complement the already given information and not to require the same knowledge to be re-entered repeatedly. This benefits the automatic maintenance of correctness across the many modeling languages used (provided the dependencies among these models could be formalized in form of constraints which is already common practice in many domains). Such an assistant would facilitate change and counter the single biggest reason for software engineering failure: the inability to propagate changes correctly and completely.

3 Fine-Tuning Transformations

In this section, we sketch in more detail the challenges that an automated assistant for change propagation should address – a challenge in which transformation plays a key role.

The classical textbook definition of a model transformation is to convert a source model into a target model where both source and target models have well-defined syntax and semantics. In the engineering of software systems, it is quite common to attribute different models (or modeling languages) to different engineers (or engineering disciplines) and their needs. Figure 2 depicts a simple pipeline where an engineer may create a model in a language most suitable for his/her work and then transform it to another model in a language most suitable for the work of another engineer who is meant to add to the knowledge provided by the first engineer. Engineering may be seen as a set of sequential or parallel transformation steps where the engineers manipulate models and transformations propagate knowledge embedded in these models for the benefit of others.

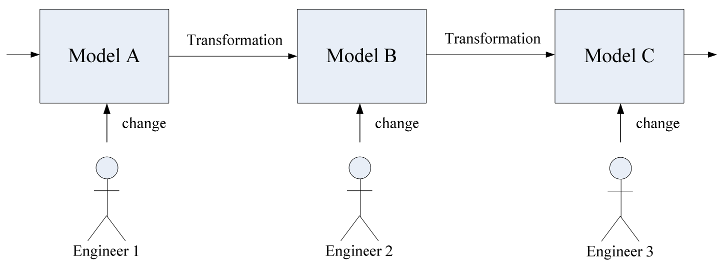

Figure 2. Transformation to Avoid Re-Entering Knowledge

In this context, transformation can have a range of different roles, such as:

1) *Transformation as translation:* translate a model from one language to another, usually with the intent of preserving the semantics of the model such that the engineer may benefit from reasoning or automations available in context of the target model (e.g., analysis or synthesis methods).
2) *Transformation as a simplification/abstraction*: simplify a source model by depicting only parts that are relevant to a concern, engineer, or discipline. The target model can be a true subset of the source model or some transformation thereof.
3) *Transformation as a merger:* combine various source models to provide a more comprehensive, integrated target model where different, separately modeled concerns are depicted together.

There are other roles of transformation of course [7], but in context of design modeling and propagating information among models and engineers, these are the most common in our experience. There is also no clear separation of the roles. Instead, transformation typically follows multiple such roles (e.g., merging and filtering go hand-in-hand). However, in all cases, *transformation propagates knowledge – knowledge that originally must have come from human engineers*. While transformation can have many roles, the main purpose of transformation is to avoid having engineers re-enter knowledge if that knowledge is already available in another model. The intent is not only to save effort but also to ensure that knowledge is propagated correctly and completely from those that create it to those that require it. Making sure that transformations are correctly chained or composed is a topic that deserves further attention [8, 9].

3.1 Transformation and Redundancies

Transformations would not be possible if models would not overlap. However, if we could compute one model entirely from another (or set of other models) then the model would not contribute new knowledge. The less knowledge a model adds, the less likely this model is going to be used during engineering as there is no value added (except for cases where model transformations are necessary to integrate tools or technologies but in the bigger context of change propagation these kinds of transformations are implementation details). Models are thus typically partially overlapping only – intentionally diverse to ensure that each model contributes new knowledge. This implies that transformation is not able to (nor meant to) compute a target model in its entirety but only fragments thereof – the parts that can be inferred from other (source) models while the remaining, new knowledge must come from the engineers or other models. The degree of overlap can vary: from no overlap (disjoint models) where no transformation is possible to partial overlaps, subsets, and complete overlaps. The “overlapping” area is either outright replication of modeling elements (physical overlap) or a re-interpretation thereof (semantic overlap). The more obscure the re-interpretation, the more complex the transformation.

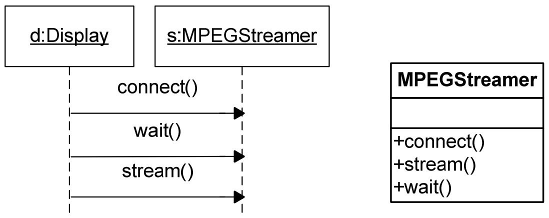

Figure 3. Semantic Overlap: A Method with the Name of the Message must be defined in the Message Receiver's Class. The method and the message are distinct model elements but they share knowledge, such as their names.

An example of a relatively simple re-interpretation of modeling elements is given in Figure 3 (based on the example introduced in the illustration in Figure 1). The message in the sequence diagram (left) is a different kind of model element than the method in the class diagram (right). Semantically, the method defines functionality whereas the message represents a specific invocation of that functionality. While these two model elements are quite distinct elements in terms of their syntax, they share knowledge: (1) the message defines the location of the method through the message receiver (e.g., message *stream()* identifies object *s* of type *Streamer*) and (2) the message defines the name of the method through the message's name. However, the message does not define the parameters of said method nor, in case of inheritance, whether the method should be in the location referenced or one of its parents. There may be heuristics for choosing among these uncertainties but again we like to point out that heuristics can be wrong and advocate against using them.

3.2 Transformation Conflict

Transformation propagates knowledge and, once propagated, this knowledge exists redundantly (in the source model and the target model). We know that redundant knowledge must be kept consistent over time. We define the source model to be consistent with the target model if all knowledge transformable from the source model to a target model is equal to the knowledge in the target model (and vice versa). If the transformation is correct then the source and target models must be consistent initially. However, what if the model changes?

If a model changes then we first need to remember all past transformations that included the changed model because the (target) models derived from these models may need to change also. This requires *traceability*, the knowledge where knowledge came from and where it was being used. See "Past Transformation" trace in Figure 4 (left).

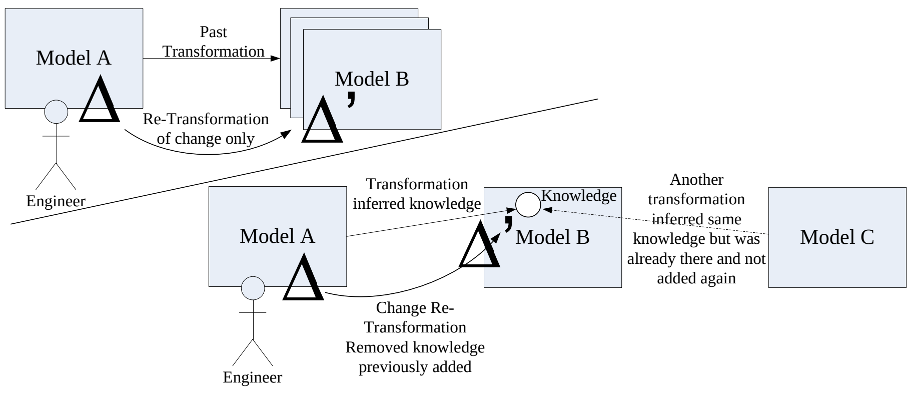
Model A
Past Transformation
Model B
Δ
Δ'
Re-Transformation of change only
Engineer
Transformation inferred knowledge
Knowledge
Another transformation inferred same knowledge but was already there and not added again
Model A
Model B
Model C
Δ
Δ'
Change Re-Transformation Removed knowledge previously added
Engineer

Figure 4. Changing a model requires updating all models to which knowledge was transformed. Transformation interactions occur when later transformations are affected by the results of earlier transformations

To support change propagation, a transformation needs to be precise in that, ideally, only the change is re-transformed and not the entire model. This is particularly important if the transformation requires manual intervention:

- during transformation by the engineer providing additional knowledge not inferable from the source model.
- after transformation in form of changes to the target model.

Here we encounter different forms of change propagation problems. Consider that one transformation method inferred some knowledge in context of a model (e.g., knowledge inferred in model B based on transformation from model A in Figure 4 right). If the same knowledge would also be derivable through another transformation (e.g., from model C) then obviously this knowledge is not created twice in the target model (model B) but rather the first transformation creates it and the second one simply terminates without creating anything. What if the source model of the first transformation changes such that it no longer infers that knowledge? If change propagation would just undo the creation of the knowledge in model B then the result would be incorrect. The knowledge should remain because there is another model that still supports it.

Such transformation interactions not only happen between automated transformations. The engineer is also a (manual) transformation engine and Figure 5 illustrates a simple dilemma that involves undesirable interactions between an engineer and a transformation. For the above illustration, we know that a method should be added to a class or its parents if a message is created. Obviously, the method should only be added if such a method does not exist but if the class also has parents then additional guidance by the engineer is necessary to specify where to add the method – to the class or to one of its parents (manual guidance). However, imagine that the message was transformed at a time where the parent did not exist. Obviously, the transformation placed the method at the only class available *at that time* – which was a correct decision that did not require human intervention. A change to, say, the message name, should then update the corresponding method (propagation). Yet, if a parent class was introduced since then, the re-transformation should not just transform the change, the name, but it should also understand that the premise of the initial transformation is no longer valid. Indeed, one might argue that this premise should already be questioned at the time the parent class is introduced.

For change propagation, the sequence of changes cannot be taken strictly. If the source and target models are manipulated by different engineers then it is largely irrelevant who made what change first. In other words, the initial transformation of the *stream()* method was correct only until the introduction of the parent class in the target model. This problem is analogous to race conditions.

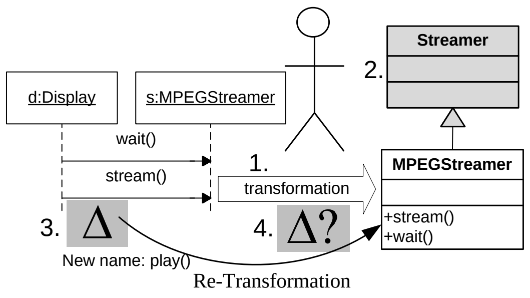

Figure 5. For Change Propagation, Transformations have an “Expiration Time Stamp”. Here, the initial transformation of the *stream()* message to the *stream()* method was correct because there was no parent. The later introduction of the parent class potentially invalidates the initial transformation.

For change propagation, we obviously require fine-grained traceability to remember where to transform to. However, we also need mechanisms for triggering re-transformation to avoid race conditions in when/how transformations happened and when/how models where changed (by transformations or engineers).

3.3 One-Directional Transformation but Bi-Directional Change Propagation

A change is neutral in terms of the transformation direction; however, often transformation is one-directional only. If an engineer likes to change some model elements and these elements were (in part) computed through transformation from other model elements then this change may cause inconsistencies. Again, we need to remember past transformation results – but this time from the perspective of the target model. Still, this problem is different from the above. If the model element we like to change was computed through a one-directional transformation method then how are we to propagate this change? We would either have to manually update both the target model(s) and the source model(s) with the same knowledge (this is not desirable) or we would need to change the source model such that the re-transformed source model causes the desired model change in the target model - a nearly impossible task in context of complex transformations (Figure 6).

The basic implication is that we need bi-directional transformation [10]. However, bi-directional transformations often do not occur in practice. Even in context of the trivial transformation from messages to methods above, it is hard to imagine how to reverse transform a method to a message. It is clear that a method should be invoked in form of messages but how many such invocations should exist or where/when they should exist is not inferable. The answer here is in partial transformation. In this simple example, many modeling tools may suggest the name of a method once a message is created. This conception of model completion is an area ripe for research on transformation [11].

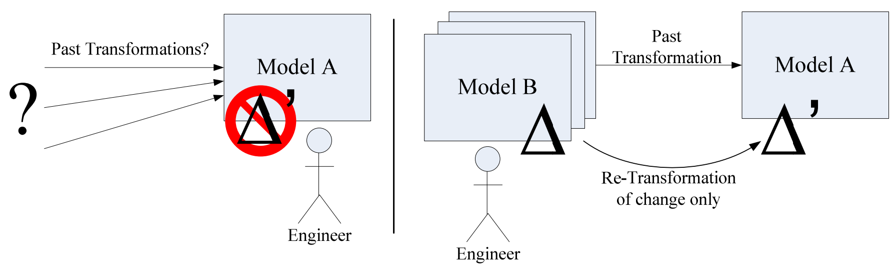

Figure 6. Change Propagation cannot be solved with One-Directional Transformation. Here: if transformation can propagate the change from model B to model A only then the engineer must either update a model A change in model B manually or attempt to change model B such that it causes the desired change in model A through transformation.

3.4 Multiple Transformation Steps and Change Propagation

Change propagation must follow redundant knowledge. If a model changes then the knowledge that was inferred from it must be re-transformed as must be the knowledge that was inferred [12, 13]. However, a change must be re-transformed only for as long as the knowledge produced during the re-transformation is different from before (Figure 7). We thus require knowledge of data differences before and after transformation [14]. Re-transformation terminates if the target model does not change.

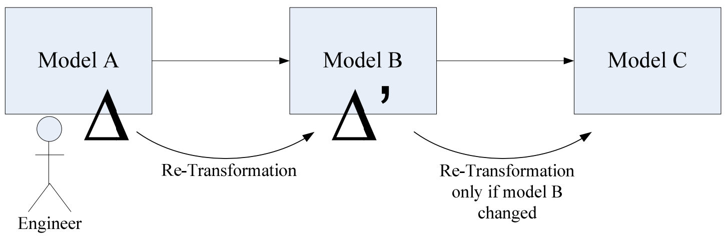

Figure 7: Re-transformation of a sequence of model transformations ends once the re-transformation does not change the targer model (here, if the re-transformation of model A to model B does not change model B then no further transformation is necessary).

Re-transforming sequentially is particularly then problematic if seen together with the problem of transformation interactions discussed above: where some but not all re-transformations unfold in the same manner

- Problem 1: what if the initial transformation required human intervention? Should re-transformation replay the human intervention? What if the source model changed in a manner were the original human intervention was no longer valid?

- Problem 2: what if the changes would trigger a different kind of transformation? Imagine that distinct transformations exist and which transformation to use depends on the contents of the source model. A change to the source model may then also change what transformation to use. It follows that re-transformation cannot blindly repeat past transformations.

3.5 Merging Transformation Results

The motivating example in Figure 1 showed that the combined changes in the class and the statechart diagrams make it possible to automatically change the sequence diagram. This is the result of combining the impact of changes from two models where each model individually would not have contained enough information for transformation to propagate the change further but together they have all information needed (not unlike parallel transformation [15] and merging [16]). Figure 8 illustrates this problem. On the surface, this problem may seem solvable by allowing multiple source models for a given transformation; thus in effect combining transformations to more complex transformations. However, in context of change propagation there may be too many transformation interactions to consider. We thus requires a different handling – one where transformations are standalone but with knowledge on how to merge results (Figure 8).

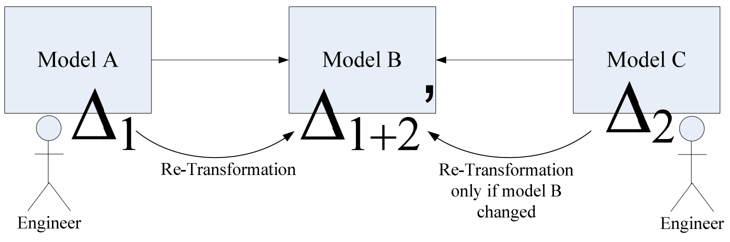

Figure 8: A Model (or its Model Elements) may sometimes be computable through the merging of multiple source models only. The dilemma: should distinct transformations be merged to single, more comprehensive transformation involving multiple sources or should transformations remain small, diverse but require explicit mechanisms for merging their results whenever necessary?

3.6 Trusting Transformation

Finally, in addition to all of the above, we have to understand that modeling implies the presence of errors (inconsistencies [17]). Indeed, a change is only necessary if the current model is no longer consistent with the engineers intention. After all, the very essence of modeling implies accepting and living with inconsistencies. Given that engineers may tolerate any number of inconsistencies, the final question is about trustworthiness of transformation (results) if we know that neither source models nor target models are complete or correct. In part, we addressed this problem in section 3.2 above when we spoke of transformations having an “expiration date.” However, should we treat transformation results differently if we know that they are based on model elements known to be contributing to inconsistencies? Works like [18, 19] are able to compute whether model elements contribute to inconsistencies. Any (subset of) transformation results that are directly or indirectly based on such “contributing” model elements have to be flagged such that the engineers are spared follow-on errors elsewhere. That is, model elements contributing to inconsistencies must identified and flagged such that engineers are aware of them in the model(s) they contributed to and in all their transformed forms.

4 Conclusions

The role of a smart assistant during change propagation is to guide the human engineer in a manner that is correct and complete. Change propagation can only be automated to the degree that 1) failure to propagate changes is observable and 2) suitable transformation methods exist to propagate knowledge. A smart assistant for change propagation thus requires the integration of consistency checking technologies and transformation technologies. Transformations are needed to move knowledge between models and consistency checking is needed for understanding when and how to transform that knowledge. The focus of this paper was specifically on the role of transformation. We discussed major transformation capabilities needed to support change propagation ranging from support for bi-directional transformation to understanding the validity of transformation results and correspondingly the need for re-transformation. We believe that transformation for change propagation is only partially automatable; hence the need for incremental transformation and the transformation of partial results (e.g., variability in form of choices and alternatives). This paper explored these and other issues, and sketched challenges and opportunities.

5 Acknowledgments

We would like to gratefully acknowledge the Austrian Science Fund (FWF) through grants P21321-N15 and M1268-N23, and the EU Marie Curie Actions – Intra European Fellowship (IEF) through project number 254965.

6 References

[1] H. Gall and M. Lanza, "Software evolution: analysis and visualization," in *Proceedings of the International Conference on Software Engineering* 2006, pp. 1055-1056.

[2] P. Tarr, H. Osher, W. Harrison, and S. M. Sutton, Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns," in *Proceedings of the 21st International Conference on Software Engineering (ICSE 21),* 1999, pp. 107-119.

[3] S. A. Bohner and R. S. Arnold, *Software Change Impact Analysis*: IEEE Computer Society Press, 1991.

[4] J. Cabot, R. Clarisó, E. Guerra, and J. d. Lara, "Analysing Graph Transformation Rules through OCL," in *1st International Conference on Theory and Practice of Model Transformations (ICMT)*, Zürich, Switzerland, June 2008, pp. 229-244.

[5] A. Egyed, "Automatically Detecting and Tracking Inconsistencies in Software Design Models," *IEEE Transactions on Software Engineering (TSE),* vol. 37, pp. 188-204 2011.

[6] J. K. A. Warmer, *The Object Constraint Language*. Reading, MA: Addison Wesley, 1999.

[7] T. Mens, K. Czarnecki, and P. V. Gorp, "04101 Discussion - A Taxonomy of Model Transformations," in *Language Engineering for Model-Driven Software Development, Dagstuhl Seminar Proceedings*, Schloss Dagstuhl, Germany, 2005.

[8] F. Heidenreich, J. Kopcsek, and U. Aßmann, "Safe Composition of Transformations," in *3rd International Conference on Theory and Practice of Model Transformations (ICMT)*, Malaga, Spain, June 2010, pp. 108-122.

[9] T. Hettel, M. Lawley, and K. Raymond, "Model Synchronisation: Definitions for Round-Trip Engineering," in *1st International Conference on Theory and Practice of Model Transformations (ICMT)*, Zürich, Switzerland, June 2008, pp. 31-45.

[10] K. Czarnecki, *et al.*, "Bidirectional Transformations: A Cross-Discipline Perspective," in *2nd International Conference on Theory and Practice of Model Transformations (ICMT)*, Zurich, Switzerland, June 2009, pp. 260-283.

[11] S. Sen, B. Baudry, and H. Vangheluwe, "Towards Domain-specific Model Editors with Automatic Model Completion," *Journal of Simulation,* vol. 86, pp. 109-126 2010.

[12] F. Jouault and M. Tisi, "Towards Incremental Execution of ATL Transformations," in *3rd International Conference on Theory and Practice of Model Transformations (ICMT)*, Malaga, Spain, June 2010, pp. 123-137.

[13] W. Shen, K. Wang, and A. Egyed, "An Efficient and Scalable Approach to Correct Class Model Refinement," *IEEE Transactions on Software Engineering (TSE),* vol. 35, pp. 515-533, 2009.

[14] Z. Hemel, D. M. Groenewegen, L. C. L. Kats, and E. Visser, "Static consistency checking of web applications with WebDSL," *Journal of Symbolic Computation,* vol. 46, pp. 150-182, 2011.

[15] A. Cicchetti, D. D. Ruscio, and A. Pierantonio, "Managing Dependent Changes in Coupled Evolution," in *2nd International Conference on Theory and Practice of Model Transformations (ICMT)*, Zürich, Switzerland, June 2009, pp. 35-51.

[16] Y. Xiong, H. Song, Z. Hu, and M. Takeichi, "Supporting Parallel Updates with Bidirectional Model Transformations," in *3rd International Conference on Theory and Practice of Model Transformations (ICMT),* Malaga, Spain, June 2010, pp. 213-228.

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

[18] A. Egyed, "Fixing Inconsistencies in UML Design Models," in *Proceedings of the 29th International Conference on Software Engineering* Minneapolis, MN, 2007, pp. 292-301.

[19] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency Management with Repair Actions," in *Proceedings of the 25th International Conference on Software Engineering (ICSE),* Portland, Oregon, USA, 2003, pp. 455-464.