Published in the Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE), Linz, Austria, September 2004, pp. forthcoming.

Instant and Incremental Transformation of Models

Sven Johann
University of Applied Sciences Mannheim
Windeckstrasse 110
68163 Mannheim, Germany
sven_johann@gmx.net

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
aegyed@ieee.org

Abstract

This paper introduces a framework for the instant and incremental transformation of changes among models. It can be configured to understand where and when changes happen in a given source model and the impact these changes have onto a given target model. It can also be configured to select translation rules as needed to update the target model. Incremental transformation is an alternative to the batch transformation and is significantly more efficient in maintaining the synchronicity among large-scale models.

1. Introduction

Transformation moves (translates) data among separately captured and maintained sources. Its role is to carry data from one source to another one and to overcome syntactic and semantic differences in how this data is represented in different sources.

Software modeling is a classical domain that captures and maintains data separately. Little data integration exists in the modeling domain. Even design methodologies, such as the UML [5], capture and maintain diagrammatic data (i.e., class, sequence, collaboration, use-case diagrams) separately. To date, batch transformation is the predominant way of sharing data across models. For example, tools share data through import/export functions. This works well if the data sharing is infrequent or the data quantity is small. Unfortunately, iterative software development (i.e., spiral model [1]) encourages changes to be frequent and industrial models tend to be large. This implies that relatively minor but frequent model changes are computationally very expensive to transform. Here, batch transformation is impractical for maintaining data synchronicity.

This paper discusses a framework for incremental transformation. Incremental transformation understands changes and their effects. It focuses only on those parts of data that are affected by changes, thus ignoring data during transformation that does not change [2]. This makes transformation computationally more efficient and less time consuming.

Our framework is the result of three implementations on four types of models. While all implementations used different transformation rules, they had in common an infrastructure that we believe to be generally applicable for incremental transformation. The infrastructure with its implementations was built for several industrial partners who validated its scalability and usability on industrial models with up to 43,000 model elements. We found that the cost of incremental transformation is small in comparison to batch transformation.

Incremental transformation is challenging because it is hard to understand the effect of changes if models differ syntactically and semantically. Our approach uses scopes, notification mechanisms, queuing, and filtering to handle this problem.

2. UML / ESCM Case Study

We will illustrate our approach on the UML to ESCM case study. ESCM [4] is a special-purpose modeling language for the embedded systems domain. It defines over 20 types of model elements such as components, receptacles, and events. For brevity, we will use a small subset of the ESCM only.

We implemented a batch and an incremental transformation technique to support the UML to ESCM transformation. Both implementations were evaluated on a range of industrial, embedded systems models to evaluate effectiveness, optimality, and scalability.

3. Should Exist and Does Exist

Batch transformation discards transformation results while incremental transformation updates them. Thus, the key difference between batch transformation and incremental transformation is in remembering previous transformation results.

Incremental transformation observes changes to the source. It then translates those parts of the source that change the target. It creates elements in the target if such elements do not exist, it modifies elements if such elements exist but have changed, and it deletes elements.

if $[shouldExist() \wedge \neg doesExist()]$ create()
else if $[shouldExist() \wedge doesExist()]$ modify()
else if $[\neg shouldExist() \wedge doesExist()]$ delete()

Figure 1. ShouldExist/DoesExist Algorithm

In principle, incremental transformation is about understanding what target elements should exist and do exist (Figure 1). If incremental transformation determines that a target element *should exist* but it currently *does not exist* then it creates the element. Otherwise, if the target element *should exist* and it *does exist* then incremental transformation modifies it. Finally, if the target element *does exist* but it *should not exist* then incremental transformation deletes it.

Incremental transformation must understand *when* changes happen and *where* changes happen. Understanding this requires the instrumentation of the source. In [3] we discussed how to instrument several commercial-off-the-shelf design tools (such as IBM Rational Rose and Matlab/Stateflow) to expose changes to their internal design models. Interested "observers", such as our incremental transformation, are then notified about changes instantly. We implemented the UML to ESCM transformation on top of IBM Rational Rose. For example, creating a UML class in Rose results in the following time-stamped UML change notifications:

New model element: 101 UML.Class
Modified model element: 100 UML.Model [ownedElements]

The first message notifies of the creation of a model element of type *Class* with ID 101. The second message tells about a change to the *ownedElements* field of an existing model element (*Model*) with ID 100. The second message is a side effect of the creation of the class in that a pre-determined model now owns the class (model's *ownedElements* field).

Change messages communicate *when* (timestamp) and *where* (unique ids of model elements and their field names) changes happen in the UML. The *shouldExist* function then computes the number and types of elements that should exist in the target based on the changes to the source.

Implementing *shouldExist* efficiently for any target model is not trivial. Only certain types of source changes cause certain types of target changes (see Figure 2). For example, only changes to UML classes and their stereotypes cause the creation of ESCM components.

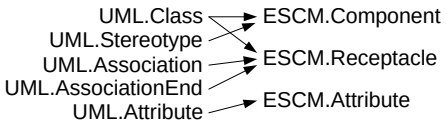

Figure 2. UML Changes relating to ESCM Changes

Thus, the two UML change notifications above affect at most ESCM components and ESCM receptacles but no other elements. Therefore, the change to the UML class triggers a call to *shouldExist* for ESCM components and it triggers another call to *shouldExist* for ESCM receptacles. This solution has several benefits: (1) it divides *shouldExist* into smaller, independent evaluation functions (instead of a single, comprehensive one) and (2) it allows for the possibility that single source elements cause the creation of multiple target elements.

For ESCM components, *shouldExist* returns true if the UML change is about an UML class with the stereotype "abstract component" or "concrete component" (a *stereotype* is a textual field owned by a class). The function has to consider that a change originates from either a stereotype or a class.

Component.shouldExist(change)
if (change.object is-a UML.Stereotype) then
  base = owner(change.object)
else if (change.object is-a UML.Class) then
  base = change.object
return (base is-a UML.Class and
    (stereotype(base) = "Abstract Component") or
    (stereotype(base) = "Concrete Component")

The implementation of *doesExist* is simple if the target elements have predictable, unique identifiers. The function *doesExist* returns true if a target element exists with a given id.

It is straightforward to compute a predictable id for target elements if they are based on unique source elements (called base elements). For example, the base element for every ESCM component is the UML class with the matching stereotype. The same UML class is never a base element for any other ESCM component. The function above thus looks for the base element first

(i.e., in case the changed element is a stereotype then the base element is the class that owns it). We then use the unique ID of the source element to compute the ID of the target element.

new model element: 102 UML.Stereotype
modified model element: 101 UML.Class [stereotype]

To create an ESCM component, we need to add a valid stereotype to the class 101 (see above change notifications). Now, *shouldExist* returns true while *doesExist* returns false (no ESCM element 101 exists). The *create* function instantiates an ESCM component (see Figure 3; left=UML; right=ESCM).

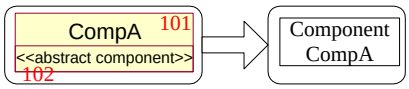

Figure 3. Creation of an ESCM Component

The *modify* and *delete* functions change an existing target element. The target element is found through the predicted unique ID. For example, modifying the name of the previously created class (101) causes the following UML change notification:

modified model element: 101 UML.Class [name]

From the UML to ESCM mapping in Figure 2 we know that this change affects either an ESCM component or an ESCM receptacle. No ESCM component needs to be created because *shouldExist* is still true and *doesExist* is true also. The ShouldExist/DoesExist algorithm in Figure 1 thus modifies the existing ESCM component. This modification includes the update of its field values.

4. Creation Scope and Update Scope

Figure 2 defined a generic mapping table that related *types* of source elements to *types* of target elements they create. For example, a change to any UML class or stereotype triggers a shouldExist/doesExist evaluation for ESCM components. We have no knowledge a-priori what instances of source elements trigger the creation of target elements.

Once a target element exists, its modification and deletion is dependent on the specific source elements that caused its creation.[1] For example, the existence of *CompA* is dependent solely on the class *instance* 101 and the stereotype *instance* 102.

Incremental transformation maintains the scope for *create* separately from the scope for *modify* and *delete*. The *creation scope* refers to types of source elements that trigger the creation of types of target elements, if changed. The *update scope* refers to instances of source elements that trigger the modification and deletion of instances of target elements, if changed.

Separating the creation scope from the update scope not only separates types from instances but it also separates scope boundaries. Typically, the update scope contains more source elements than the creation scope because the existence of a target element is typically computable with a subset of the knowledge.

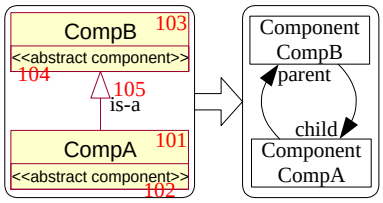

Figure 4. Creation of an ESCM Receptacle

For example, if we create a second UML class with the name *CompB* and the "abstract component" stereotype then a second ESCM component is created (see Figure 4; left=UML; right=ESCM). Adding an inheritance relationship between the two classes does not affect the creation of any ESCM element but it does modify both existing ones. That is, component *CompA* becomes a child of component *CompB*.

Therefore, the creation scope for ESCM components is UML classes and UML stereotypes (as depicted in Figure 2). UML Generalizations (inheritance relationships) are not part of this creation scope. However, the update scope for UML components is the specific classes, stereotypes, and generalizations that define its creation and its fields. For example, the update scope of component *CompA* is UML class *CompA* (101), UML stereotype 102, and UML generalization *is-a* (105) but not the class *CompB* (103). The update scope is defined during translation.

Some source elements are part of the update scope of multiple target elements. For example, the UML generalization *is-a* (105) is part of the update scope for both components *CompA* and *CompB*. A change in the generalization affects the *children* field of *CompB* and it affects the *parent* field of *CompA*.

[1] We use the term instances and elements synonymously. Certain instances are related in their type (e.g., two *instances* of the *type* UML class)

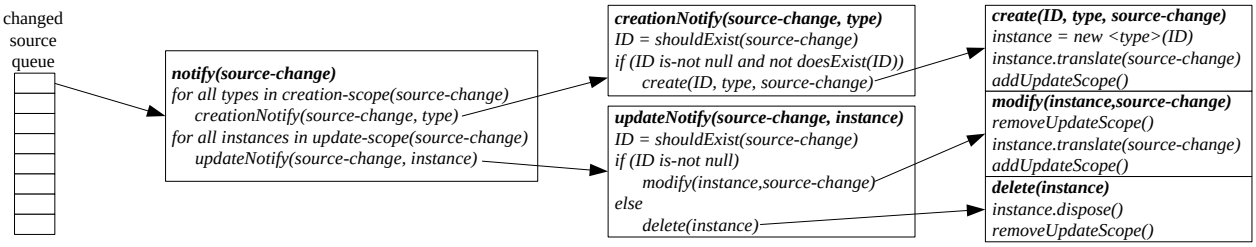

Figure 5. Basic Incremental Transformation

5. Infrastructure

Figure 5 depicts the basic infrastructure for incremental transformation with scopes. UML change notifications trigger calls to the *notify* function. The *notify* function first calls *creationNotify* for every affected target type (e.g., a change to an UML class may create an ESCM component). The function then calls *updateNotify* for every affected target instance (e.g., a change to the class 103 may change the ESCM component *CompB*).

The *creationNotify* function computes *shouldExist* and *doesExist*. To optimize the approach, the *shouldExist* function returns the predicted unique ID of the base element that should exist. The *doesExist* function takes the predicted ID and returns the actual target instance; or it returns null if it does not exist. The *create* function is called in accordance to the ShouldExist/DoesExist algorithm discussed earlier. It first creates an instance of the required type and predicted ID. It then calls translate to set the field values (e.g., *name*, *parent*, *children*).

The *updateNotify* function calls *modify* if *shouldExist* returns an ID (this is equivalent to returning *true*). Otherwise, the *delete* function is called. Both functions are called in accordance to the ShouldExist/DoesExist algorithm because the target elements do exist if there is a defined update scope (i.e., update scope is added in *create* and removed in *delete*). The *modify* function removes and adds the update scope because a change to some target elements also affects their scope.

The role of translate is two-fold: First, it investigates source elements to compute field values (properties) for individual target elements. Second, it computes the update scope for these target elements. Thus, one translate function is needed per target element type.

6. CONCLUSIONS

This paper presented an overview of a framework for the incremental transformation of models. The framework supported the transformation of changes from a source model to a target model. The goal of the framework was to minimize unnecessary transformation by only transforming data that changed. The presented framework is a simplification of the actual framework in that we omitted details. For example, it was not discussed how semantic changes are treated. Those are simple changes in the source model that cause a variety of ripple effects among multiple/many target elements. Indeed, our extended framework supports this ripple effect which will be presented in a follow-on paper.

The framework was implemented three times and it was validated on several, large-scale industrial models with up to 43,000 model elements. The large-scale nature of the validation models was necessary because they are the main motivation for incremental transformation (i.e., batch transformation would suffice for small-scale models). The validation determined that incremental transformation comes with a slight performance penalty initially (during initial transformation) but is computationally very effective thereafter (with every change).

7. REFERENCES

[1] Boehm B., Egyed A., Kwan J., and Madachy R.: Using the WinWin Spiral Model: A Case Study. *IEEE Computer*, 1998, 33-44.

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

[3] Egyed, A. and Balzer, R. "Unfriendly COTS Integration - Instrumentation and Interfaces for Improved Plugability," *Proceedings of the 16th IEEE Intern. Conference on Automated Software Engineering*, San Diego, USA, Nov. 2001.

[4] Roll, W.: "Towards Model-Based and CCM-Based Applications for Real-Time Systems ," *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Hakodate, Hokkaido, Japan , May 2003, pp.75-82.

[5] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.