Traceability-based Change Management in Operational Mappings

Ivan Kurtev[1], Matthijs Dee[1], Arda Goknil[1], Klaas van den Berg[1]

[1] Software Engineering Group, University of Twente
7500 AE Enschede, the Netherlands
{kurtev, K.G.van.den.Berg, goknila}@ewi.utwente.nl
m.r.dee@student.utwente.nl

Abstract. This paper describes an approach for the analysis of changes in model transformations in the Model Driven Architecture (MDA). Models should be amenable to changes in user requirements and technological platforms. Impact analysis of changes can be based on traceability of model elements. We propose a model for generating trace links between model elements and study scenarios for changes in source models and how to identify the impacted elements in the target model.

1 Introduction

Change management is a prerequisite for high-quality software development. Changes may be caused by changing user requirements and business goals or be induced by changes in implementation technologies. Software architectures must be designed such that they can evolve to cope with these changes. The Model Driven Engineering (MDE) approach aims at providing stable models amenable to changes [14]. An analysis of the impact of changes is necessary for a cost effective software development [3]. The number of affected modules or elements is a response measure for the quality attribute *modifiability* in software architectural design [4]. Such analysis can be based on dependency traces between elements in the architectural design and other software artifacts.

In MDE, models are manipulated via model transformations. Transformations are usually sequentially applied until a model with enough details is obtained. A change in one of the source models causes changes in all the models obtained as products in the transformation chain. There are two options for performing these changes: (1) executing the transformations again on the whole modified model and (2) propagating the changes incrementally by transforming only the changed source elements. In the latter case we need an incremental model transformation mechanism. It is clear that in the case of large models the incremental transformation approach may be more efficient.

In order to perform change impact analysis we need to trace changes in a source model element to required changes in the target model elements. We utilize trace information created during the transformation execution. It provides a set of traces

that relate source and target elements created by a given transformation rule. The necessity of such traces points to the requirement for an important quality property of model transformation languages and execution engines: *traceability*.

Traceability is defined as the degree to which a relationship can be established between two or more products of the development process [10]. In the context of model transformations the products are models and their model elements. Traceability is an optional requirement in the QVT Request for Proposals (RFP) issued by OMG [15]. QVT specification describes three model transformation languages: Relations, Core, and Operational Mappings. In the Relations and Operational Mappings languages, traces are created automatically and remain transparent to the user. In the Core Language, a trace class is specified explicitly for each transformation mapping.

In this paper we study the possibility for using incremental model transformations written in the QVT language Operational Mappings. We use the transformation engine provided by Borland Together Architect 2006 for Eclipse to execute transformations and to experiment with the trace information. We evaluate the possibility to use the traces generated by Together Architect as a side product of a transformation execution. We classify change cases and analyze them in the context of incremental model transformations.

The paper is structured as follows. In Section 2, we describe our approach to generation of traces. Section 3 gives a conceptual framework for change impact analysis. Section 4 discusses possibilities and obstacles in implementing the identified changes. Section 5 describes related work and Section 6 concludes the paper.

2 Traceability in Operational Mappings Language

We generalize the concept of traceability by means of a traceability pattern (see Figure 1).

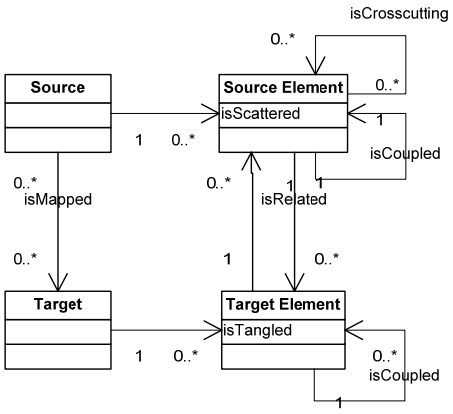

Figure 1. Traceability Pattern

In this pattern, we have dependency relations between elements in the source and elements in the target. We use here the general terms *source* and *target* to denote two consecutive levels. We distinguish between intra-level and inter-level dependency

relations. Intra-level relations denote couplings between elements at a certain level (model). Examples of intra-level relations are generalization between classes, aggregation, etc. Inter-level relations relate elements at different level of abstraction. For example, an element in a model is refined to a set of elements in another model at a lower level of abstraction. In this case we have a *refinement* inter-level relation. In the case of model transformations inter-level relations are the traces derived during the transformation of source elements to target elements. For the purpose of change impact analysis both types of dependencies should be taken into account.

We may distinguish several cases of mappings between source and target: 1-to-1, 1-to-many, many-to-1, and many-to-many. This can be represented in a dependency graph, as shown in Figure 2.

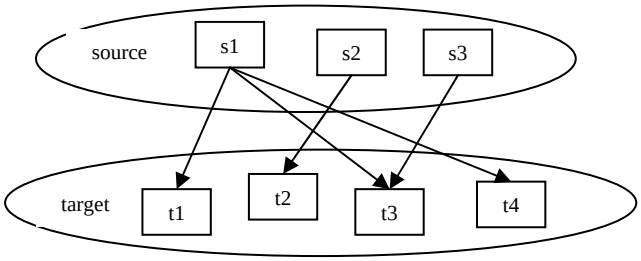

Figure 2. Mapping between elements at different levels of abstraction (s1, s2, s3 at source; t1, t2, t3 and t4 at target)

In general, a transformation language provides different ways for creating and maintaining the traces between source and target elements. Operational Mappings language uses automatic creation of the traces. They can be used during the transformation execution by invoking different forms of the *resolve* function. The QVT specification does not impose any constraints on the exact structure of the traces and their lifetime. This is considered an implementation specific issue.

The engine implemented in Together Architect provides an option for creating persistent traces, i.e. traces that are saved after the transformation execution. Keeping such traces in a form accessible to programs is essential for the change impact analysis. Unfortunately, Together Architect only provides a browsing mechanism to inspect the traces. They cannot be manipulated programmatically. This is the reason for implementing our own traceability mechanism that produces persistent traces in the form of a model.

In our approach traces are instances of a simple model. Every trace is associated to a rule and refers to the sets of source and target elements used by that rule. Formally traces have the following structure:

```
t([s1, …, sn], [t1, …, tm], r)
```

This is interpreted in the following way: trace *t* is derived from the execution of rule *r* on source model elements *s1,…, sn* that results in the creation of target model elements *t1,…, tm*. Since a rule may match multiple tuples in the source model it is clear that multiple traces may exist per single transformation rule.

The set of traces form a model generated after the execution of a transformation. The generation is done by inserting code in every transformation rule in a given transformation definition. The automation of this process is beyond the scope of this paper. More information may be found in [12]. The model with traces conforms to a trace metamodel. It should be noted that this metamodel is different from the traceability pattern in Fig. 1. The trace metamodel describes in details the relations between source and target, and between source element and target element.

3 Traceability-based Change Impact Analysis

We present a classification of change cases and analyze them in terms of traces, rules to be executed, and elements affected in the source and target models. Due to a lack of space we give only three cases. The complete set of cases is available in [8].

3.1 Notation

In the remainder of this section models will be used to show the inter-level trace dependencies between source and target model elements. The legend for these models is shown in Figure 3.

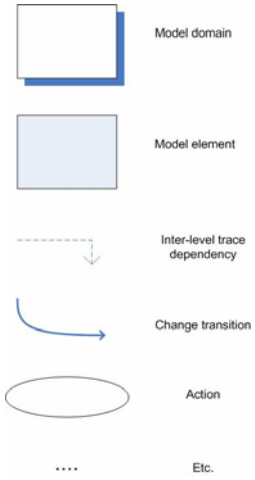

Figure 3 Visual notation used in the examples

We assume that models are sets of elements. We denote models in the following way:

Source model: $S = \{s_1, s_2, \ldots, s_n\}$; Target model: $T = \{t_1, t_2, \ldots, t_m\}$

The target model is generated by an execution of a transformation definition R that consists of transformation rules: $R = \{r_1, r_2, \ldots, r_k\}$

During the execution of the transformation traces are created. These traces hold source and target model elements and the transformation rule which created the trace:

Set of traces: $I = \{i_1, i_2, \ldots, i_l\}$; Trace: $i_1 = (s_1, t_1, r_1)$

We need to indicate changes being made to a model. In the following sections we limit ourselves to two change types: *update* and *delete*. In some cases we need to denote a change without specifying the type. We use the wildcard (*) in this case.

Change-types : $C = \{c_u, c_d, c_*\}$, where *u* stands for *update* and *d* for *delete.*

With these sets we can define a method that takes a change, a model, and the impacted elements, and creates the new desired model:

`change([change-type],[model],[element(s)]) = [model]'`

where:

`[model]` is a model, `element(s)` is a set of model elements affected by the change, and `[change-type]` is a change-type from `C`.

When applying a change to a source model, we need to keep the target model consistent with this change. This consistency is preserved with the implementation of the same change on the target model and thus the impacted element(s). The unidirectional transformation from `S` to `T` is indicated by `creates`. We assume that the target model is not changed in the meantime (e.g. edited manually) and all the changes are driven by changes in the source model.

3.2 Example Case: one-to-one Mappings

This case has two sub-cases:

- a single source model element is mapped to a single target model element with one transformation rule;
- multiple transformation rules create the target model element;

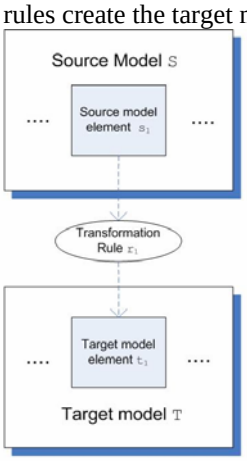

Figure 4 One to one inter-level trace dependency created with one transformation rule

Figure 4 shows a one-to-one trace example where the element s_1 is transformed to the element t_1 by executing rule r_1: $r_1(s_1) = t_1$. The only trace created holds source element s_1, target element t_1 and transformation rule r_1.

The data contained in the source element s_1 is mapped to target element t_1. This indicates the lowest level of granularity on which this information is used to create source elements' data.

When changing s_1 the impacted elements are found by following the trace leading to the target model element t_1. The trace indicates the usage of the transformation rule r_1. The change on s_1 must be propagated to the impacted target model element. The following facts are known:

S creates T, $i = (s_1, t_1, r_1)$, $Change(c_*, S, \{s_1\}) = S'$, $Change(c_*, T, \{t_1\}) = T'$

The desired implementations of the change types are schematically shown in Figure 5 and Figure 6.

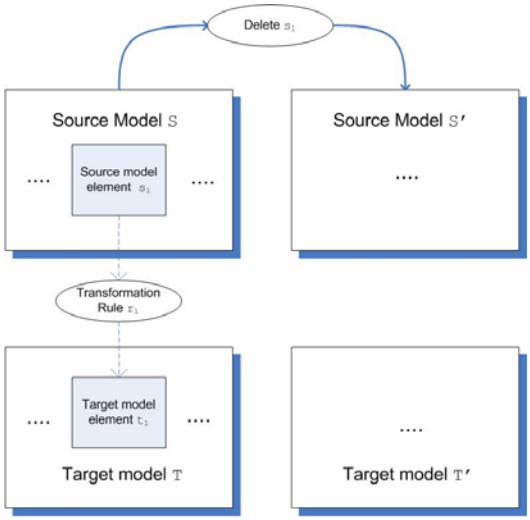

Figure 5 Delete change in one to one mapping

The change types infer two possible scenarios:

- *Delete the source element*

  Deleting s_1 from S implies the deletion of t_1 from T. This results in the following change implementations, which creates the new source and target models:

  $change(c_d, S, \{s_1\}) = S \setminus \{s_1\}$

  $change(c_d, T, \{t_1\}) = T \setminus \{t_1\}$

- *Update the source element*

  Updating s_1 from S implies the re-execution of transformation rule r_1 and replacing t_1 with the newly created target model element. This results in the following change implementation:

  $change(c_u, S, \{s_1\}) = S \setminus \{s_1\} \cup \{s_1'\}$ where s_1' is manual input

  $change(c_u, T, \{t_1\}) = T \setminus \{t_1\} \cup \{t_1'\}$ where $t_1' = r_1(s_1')$

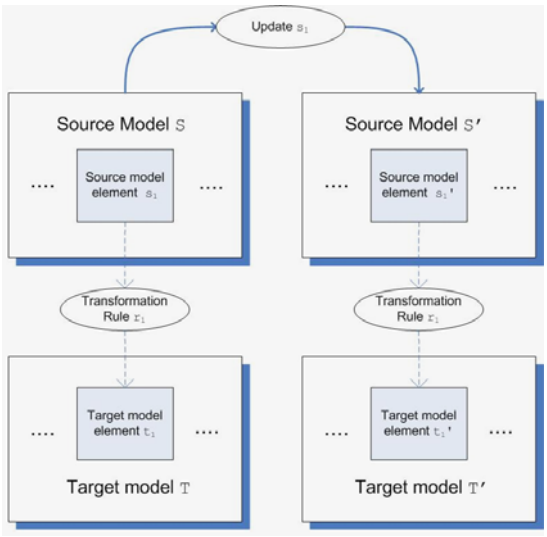

Figure 6 Update change in one-to-one mapping

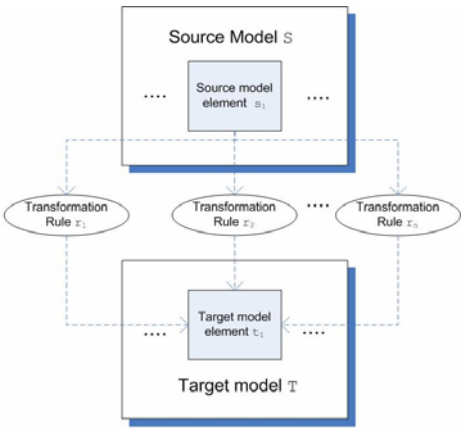

Figure 7 One to one inter-level trace dependencies created with multiple transformation rules

The previous example can be extended to a sub-case where the source element is mapped to one target element by using of multiple transformation rules. The difference between the two cases is the number of traces created during the execution of the transformation and the number of rules executed to create the target element. This is schematically shown in Figure 7.

In this example a trace is created for every transformation rule execution. We assume that there is a fixed order in which these transformation rules are executed (reflecting the imperative nature of Operational Mappings). Because of this assumption we get a sequential list of transformation rules $\{r_1, r_2, \ldots, r_n\}$.

When the traces are collected, a list of traces with the following structure is created:

$$\{(s_1, t_1, r_1), (s_1, t_1, r_2), \ldots, (s_1, t_1, r_n)\}$$

Here we consider only the case of update of the source element. Updating s_1 from S implies the re-execution of transformation rules $\{r_1, r_2, \ldots, r_n\}$ and replacing t_1 with the newly created target model element. This results in the following change implementation:

$change(c_u, S, \{s_1\}) = S \setminus \{s_1\} \cup \{s_1'\}$ where s_1' is manual input

$change(c_u, T, \{t_1\}) = T \setminus \{t_1\} \cup \{t_1'\}$ where $t_1' = r_1(s_1'); r_2(s_1'); \ldots; r_n(s_1');$

4 Discussion

In this paper we limit ourselves only to performing impact analysis. The next step is to implement the changes. We give an overview of possibilities and the potential obstacles.

In general, there are two ways to implement the required changes: generate a function that implements the change (a kind of "patch") and re-execute rules.

A patch function should be generated based on the logic of the transformation rules responsible for changes. A major problem is the imperative nature of operational mappings rules. It is not clear how the required functionality is extracted from a rule and how it is connected to its original context. We may hypothesize that using an imperative transformation language is not the best option for this scenario. The applicability of a language based on another paradigm (e.g. declarative, graph transformation-based) needs a further study.

The second option is to re-execute a rule. In the worst case this option may completely fail. In the current implementation of Borland together it is not possible to execute an arbitrary rule. Furthermore, the engine does not provide any customizable features. Apart from that, we also have a problem of granularity. Consider a case in which only an attribute value needs to be changed. This is done by executing of an assignment part of a rule. The smallest executable modules in Operational Mappings language are helpers and mappings. We cannot execute parts of a mapping. Instead, we need to execute the whole mapping. This, however, may lead to execution of other mappings invoked by the required one. There is no way to prevent this without knowing the internal implementation details of the transformation engine.

The current version of the approach to impact analysis has three major limitations:

- *Lack of intra-level reasoning.* When a model element is changed this may lead to changes in other elements in the same model due to the dependencies imposed by the language semantics. We call such dependencies intra-level dependencies. For example, if a class is deleted then its attributes must also be deleted. Another example is the effect of deleting a super class. This automatically changes the set of attributes of the specializations of that class. Both examples are related to the semantics of the modeling language being

used. The lack of standard way to define language semantics prevents us from performing generic intra-level reasoning. It should be done per every modeling language thus limiting the generality of the impact analysis framework.

- *Changes in guards are not considered*. Our tracing model keeps track on the relations between source and target model elements established during transformation execution. However, a change may also be caused by changing values that are checked by mapping guards. A source element may not be mapped and therefore no trace for it is available. Yet, this element may influence the evaluation of the guard in a mapping. Changing such an element will have no consequences since no trace is available for it. The only way to overcome this limitation is to enhance the trace model by including also the elements participating in the guard expressions.
- *Additions of elements are not handled.* Addition is problematic due to the same reason mentioned in the previous case: no trace is available for the newly added element. To handle the change, we need to identify the applicable mappings. However, due to the imperative nature of the language, such a mapping may be invoked in the context of another one and may invoke other mappings. Again, identifying the functionality for the change implementation is problematic.

5 Related Work

The topic of incremental model transformations is studied in [9] and [11]. Hearnden et al. [9] studies how continuous handling of changes in a source model may be done with the Tefkat language [13]. Their analysis is based on a detailed knowledge of the execution semantics of the language. The implementation of the changes extends the engine with additional structures for keeping intermediate execution context information. In this paper we study another language with different semantics. It was not possible to come with implementation of changes since we could not extend the transformation engine for Operational Mappings.

The authors of [11] propose writing model transformations in a style that is based on anticipating changes. Transformations are event-driven. Events are adding and deleting model elements. Events trigger transformation rules. This paradigm is different from the current model transformation languages that are not designed to anticipate changes in the models.

In [1], a number of events and change actions have been defined as part of an operational semantics of traceability. These events and change actions could be the start of a change impact analysis as described in this paper.

In [16], the generation of traceability links is discussed, especially between requirements and the object model, and between requirements. This corresponds to the inter-level dependencies and the intra-level dependencies as described in this paper. Similarly in [5], traceability links are retrieved between UML and target models including one-to-many relations. In the current paper, we focused on generated trace relations as part of QVT transformations. An event-based approach to traceability is

described in [6]. In this approach, change is handled by means of event notification and propagation of changes using traces between artifacts.

6 Conclusion

In this paper, we described an approach for the analysis of change in model transformations. We defined a traceability model for generating traces between source and target model elements. We analyzed the change impact in case of modifications of the source model. Several change scenarios were analyzed.

The specification of transformation rules and the tracing information of the execution of these rules can be used to generate dependency graphs at model level and at metamodel level. These dependency graphs are helpful in identifying which rule applications need to be re-executed in which order when there are changes to the source model. These are the key issues to implement incremental model transformations.

Our approach needs further elaboration and has to be validated in empirical case studies. Moreover, the derivation of dependencies and its analysis should be supported by tools in order to scale to industrial projects.

Acknowledgement

This work is performed in the context of AOSD-Europe Project IST-2-004349-NoE [2], the ESI Project Darwin [7] and the Jacquard/NWO QuadREAD Project.

References

1. Aizenbud-Reshef, N., Paige, R.F., Rubin, J., Shaham-Gafni, Y. and Kolovos, D.S. (2005). Operational Semantics for Traceability. In ECMDA-FA Traceability Workshop, Nuremberg, Germany
2. AOSD-Europe (2005). *AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation.* Retrieved May, 2005, from http://www.aosd-europe.net/documents/d9Ont.pdf
3. Arnold, R. S., & Bohner, S. A. (1993). Impact analysis - towards a framework for comparison. Paper presented at the Conference on Software Maintenance.
4. Bass, L., Clements, P., & Kazman, R. (2003). Software architecture in practice (2nd ed.). Boston: Addison-Wesley
5. Berg, K. van den & Conejero, J. (2005). Disentangling crosscutting in AOSD: a conceptual framework, in *Second Edition of European Interactive Workshop on Aspects in Software*, Brussels, Belgium
6. Cleland-Huang, J., C.K. Chang, and M. Christensen (2003). Event-based traceability for managing evolutionary change. IEEE Transactions on Software Engineering 29(9) pp. 796-810
7. Darwin (2005). Designing Highly Evolvable System Architectures. Retrieved March 13, 2006 from http://www.esi.nl/site/projects/darwin.html

8. Dee, M. (2007). Traceability-based change impact analysis in MDA. MSc Thesis, University of Twente, Dept. Computer Science.
9. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. MoDELS 2006, pp. 321-335
10. IEEE (1990). IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. Institute of Electrical and Electronics Engineers, New York
11. Johann, S. and Egyed, A. "Instant and Incremental Transformation of Models," Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE), Linz, Austria, September 2004, pp. 362-365
12. Kurtev, I., Berg, K. v. d., & Jouault, F. (2006). Evaluation of rule-based modularization in model transformation languages illustrated with ATL. Paper presented at the 21st Annual ACM Symposium on Applied Computing, Bourgogne University, Dijon, France.
13. Lawley, M., Steel, J. Practical Declarative Model Transformation with Tefkat. MoDELS Satellite Events 2005. pp. 139-150
14. MDA (2003). MDA Guide Version 1.0.1, document number omg/2003-06-01
15. OMG (2002). Request for Proposal: MOF 2.0 Query/Views/Transformations RFP, OMG document ad/2002-04-10
16. Spanoudakis G., Zisman A., Perez-Minana E., Krause P. (2004). Rule-Based Generation of Requirements Traceability Relations , Journal of Systems and Software, Vol 72(2), pp 105-127