Scenarios of Traceability in Model to Text Transformations

Gøran K. Olsen and Jon Oldevik

SINTEF Information and Communication Technology Forskningsveien 1,
0373 Oslo, Norway
{goran.k.olsen,jon.oldevik}@sintef.no

Abstract. The challenges of managing change in model-driven development are addressed by traceability mechanisms for model to text transformations. A traceability model, tailored for representing trace information between models and generated code, provides the basis for visualisation and analysis of the relationships between models and code. Usage scenarios for traceability are discussed and illustrated by our traceability implementation.

1 Introduction

Model to text transformation is one of several vital steps in model-driven development (MDD), which makes it possible to generate an extensive amount of code from models. This automation can reduce the development time and increase the quality of the code, but it also introduces some new challenges that must be addressed.

Often, the people writing transformation specifications will be different from the engineers developing the system. In this way, vital details required for understanding the systems are hidden from the engineers within the transformations. This may be a convenient way of separating the concerns of different actors in the development process. On the other hand, it may also hinder sufficient understanding of the system on the part of the engineer. One way of solving this is letting the engineer examine the design models, the transformation specifications and the generated code. This may, however, not be desirable since the engineer may be unfamiliar with the transformation language. It also reveals details that are supposed to be concealed. An alternative approach is to establish links between representations of the design artefacts and the generated code that have semantics with the necessary information for the engineers. Model to text transformations enable implicit or explicit creation of these links.

Manual updates of generated code are often required in the development process. In complex systems it can be difficult to localize the places to update, and there might also be restrictions on where changes are allowed. Traceability information can be used to ease this task.

In this paper, we describe how traceability and traceability links can be used to support the development of systems. We explain model to text traceability and how this is implemented in MOFScript [1]. Several usages of MOFScript-specific trace links are described in usage scenarios.

D.H. Akehurst, R. Vogel, and R.F. Paige (Eds.): ECMDA-FA 2007, LNCS 4530, pp. 144–156, 2007.
© Springer-Verlag Berlin Heidelberg 2007

2 Traceability

One of the main challenges in MDD is the management of relations between different artefacts produced in the development process. As systems become more complex, the number of artefacts is increasing. Furthermore, the artefacts are often generated. Therefore, trace links are needed to fully understand the many dependencies that exist between the different artefacts.

In the IEEE Standard Glossary of Software Engineering Terminology [2] traceability is defined as:

"The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match".

2.1 Establishment of Trace Links

In the past, trace links have mostly been established manually by the different persons involved in the development process, for instance by creating trace links between word documents and use-case model elements. This task has been known as difficult, time consuming, and very often a source to errors both when it comes to the establishment of new links and keeping the existing links updated and consistent [3].

Following an MDD approach and utilizing model transformations makes it possible to generate these trace links explicitly or implicitly in the transformation specification. By implicit, we mean that some transformation tool, e.g. MOFScript, populates a trace model automatically when a transformation is executed. By explicit, we mean that additional trace code must be inserted into the transformation. This can be achieved in two ways; by writing the trace code each time or running a higher order transformation on the transformation model. The latter approach is used in the Atlas Transformation Language (ATL) [4]. The final adopted OMG standard MOF Models to Text Transformation Language also requires that the ability to explicitly create trace blocks in the code is present [5].

Storing the established trace links can be done in two ways according to Kolovos et al. [6], either by embedding them in the models or storing them externally in a separate new model. The first approach gives a human-friendly view of the trace links, but it only supports trace links between elements in the same model. The external approach has the advantage of having the trace information separated from the model and therefore avoids polluting the models.

2.2 Traceability on Different Abstraction Levels

Trace links can in theory be established between all artefacts in a system development project, for instance between requirement documents and use-case diagrams, use-case diagrams and test cases or domain and platform independent models (PIM), elements in the PIM and platform specific models (PSM), and between the PSM and generated text (e.g., code and documentation).

All these trace links are required to provide end-to-end traceability. End-to-end traceability enables a number of different analyses that can be preformed on the

system, e.g. checking that a requirement is fulfilled in the implementation by following the trace links from a requirement via the PIM and the PSM to code, known as coverage analysis [7].

The trace links required to provide end-to-end traceability are intermediate and can also make the basis for useful functionality and analyses. In this paper we present several different usages of model to text traceability links.

2.3 Different Trace Link Classifications

The simplest trace link is one without any type specification other than link; it only contains references to one source and one target element which optionally can be contained in another element. According to [7-9] this may be insufficient for many projects. Hence, several different trace link classifications have been proposed.

Some examples of trace types are: The trace type *manual*, which is a trace link established manually in the trace model. The trace type *automatic* is created by a tool, and the trace type *transformation* means that the trace link is between a source and a target in a transformation. In a model to text transformation this could be from a model reference to a text segment. The trace type *dependency* is between two artefacts that are dependent of each other, and the trace type *verifies* means that one artefact verifies another (e.g. a test implementation) may verify a requirement.

2.4 Trace Link Usage

The reason to create and update traceability links is that the links can be used to support and document the development process. The information can be used in several ways, but the most obvious scenario is simple *trace inspection*. Through trace inspection it is possible to browse the trace information and get insight in how the different artefacts are connected. This is becoming more useful as an increasing number of artefacts are generated automatically from model to model and model to text transformations. The simple browsing can also be extended with additional functionality as explained in the section 0.

Walderhaug et al.[7] and Ramesh et al.[9] describe several different trace analysis scenarios:

- **Change impact analysis:** Change impact analysis is used to determine the impact a change to an artefact will have on other artefacts.
- **Coverage analysis:** Through coverage analysis, the trace user can determine the degree to which some artefacts of the system are followed up by other artefacts in the system.
- **Orphan analysis:** Orphan analysis is used to find artefacts that are orphaned with respect to some specified trace relations.

In the following section, we address traceability in model to text transformations and look at how different traceability scenarios can be provided by the traceability support in MOFScript.

3 Model to Text Traceability

For traceability to be useful, we need the ability to trace artefacts through the lifecycle of the software development process, from requirement documents to model elements and from model elements to textual artefacts such as code. The steps required to move from one level to the other are often automated by transformations. In this process, the transformation tools should be able to produce trace links.

Several model to text languages exist, and some of them have support for traceability. In the MOF Models to Text Standard [5], traceability is defined to be explicitly created by the use of a trace block inserted into the code, as illustrated below.

```
[trace(c.id()+ '_definition') ]
   class [c.name/]
   {
     // Constructor
    [c.name/]()
    {
      [protected('user_code')]
      ; user code
      [/protected]
    }
   }
[/trace]
```

This approach provides user-defined blocks that represent a trace to the code generated by the block. This is specifically useful for adding traces to parts of the code that are not easily automated. A drawback of the approach is a cluttering of the transformation code. A complementary approach, as taken in MOFScript, is to automate the generation of traces based solely on model element references.

3.1 Traceability in MOFScript

MOFScript is a model to text transformation tool and language. It can be used to generate text from EMF based models. The transformation implementation contains references to model elements that should be substituted in the generated text.

The references to model elements are the basis of MOFScript traceability. Any reference to a model element that is used to produce text output, results in a trace between that element and the target text file. The granularity is from model element to line and column in the text file [10].

```
uml.Class::main(){
    file(self.name+".java")
    'package 'packageName';\n
     import java.util.*;\n'
     self.visibility' class ' self.name'{
'
  self.ownedAttribute->forEach(p:uml.Property | p.association = null ){
    '   ' p.visibility' ' p.type.name' _' p.name';\n'

  }
  self.ownedAttribute->forEach(p:uml.Property | p.association !=null ){
  '// Association: 'p.name':'p.type.name'('p.lower '..'p.upper')'
  '\t' p.visibility' HashMap<'p.type.name', 'p.name '>_'
         p.name.toLower()';\n'
  }
```

The above transformation code generates the beginning of a Java class file where the references are fetched from the model. If the class property *visibility* is set to protected, "protected" will be written to the file instead of `self.visibility`. We will use the example model in Fig. 1 to illustrate the traceability support.

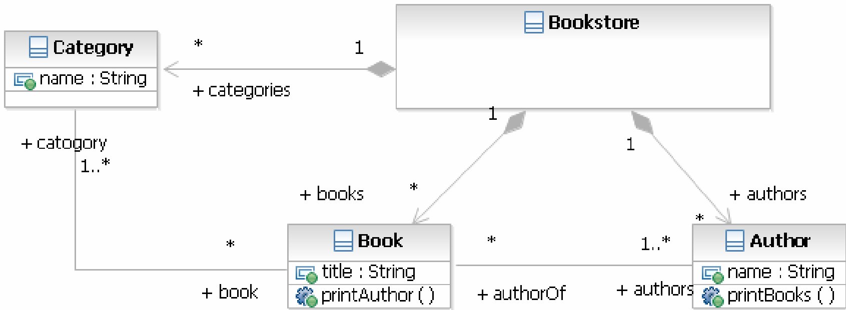

Fig. 1. Bookstore Example Model

Given the model in Fig. 1, an execution of the transformation will generate the following Java source code for the class Book.

```
package org.sintef.no;
import java.util.HashMap;

public class Book {
   private String _title ;
// Association: authors:Author(1..-1)
    protected HashMap<String, Author>_authors;
// Association: category:Category(1..-1)
    protected HashMap<String, Category>_category;
}
```

Each reference is substituted with the model element's value and a trace is created, linking the element and the code segment. The link is stored in a traceability model, an instance of MOFScript's traceability metamodel.

3.2 The Traceability Metamodel

The traceability metamodel in MOFScript was described in detail in [10]. Since then, it has been slightly modified during the implementation of the traceability support. Fig. 2 shows its concepts.

The *TraceModel* is the root of the model and contains traces, files and model element references. A *File* contains one or more blocks, which in turn contains a set of traceable segments. A *TraceableSegment* defines a position and length within a block in a file. A *Trace* references an originating model element and the segment to which it traces. The *Block* defines the positioning of the block within the file. Furthermore, a block is either protected or unprotected. A *protected block* represents an unchangeable part of a file, which is not meant to be modified by users. Conversely, an *unprotected block* represents a part of the file that is meant to be modified by the user. This could for example be the body of a method.

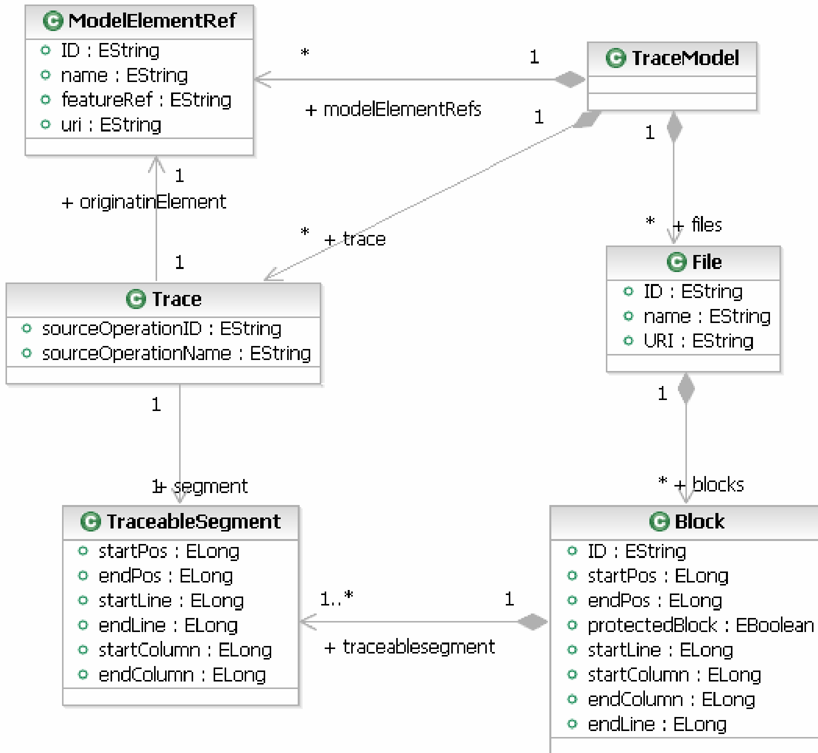

Fig. 2. Traceability Metamodel

When a transformation is executed, the MOFScript runtime populates an instance of the traceability metamodel, which is an ecore metamodel. This results in a traceability model. Fig. 3 shows an editor view of the traceability model generated from the Bookstore example model in Fig. 1.

In the traceability model in Fig. 3, *Book.java* contains, among other things, a block with id 3. In the property view we can see where the block starts and ends, and that it is a protected block. This means that editing in this area is not allowed (changes in the code will not be preserved if the file is generated again). The traceable segments represent the references that are used in the file and hold information about start and end position.

Unprotected Blocks. Setting the blocks' *protected block* property to "true" is the default behaviour of the trace generation. However, often it is required that the code is edited manually. To cope with this MOFScript supports the notion of unprotected blocks. These blocks are created with the use of the *unprotect* keyword in the transformation code, as illustrated in the transformation code for operations below.

```
self.ownedOperation->forEach(o:uml.Operation){
        '\n  'o.visibility' void ' o.name'(){'
        unprotect{
        '     //User code here for operation'
        }
        '   }\n'
}
```

The resulting code, shown below, represents the unprotected block as comments containing a *#BlockStart* and a *#BlockEnd* and an identifier for the source model element.

```
public void printAuthor(){
//#BlockStart number=4 id=_MeMJULEPEdu-Vepu7rgPLg
      //User code here for operation
//#BlockEnd number=4
}
```

Between the block comments, the user can insert or remove code, and the changes will be preserved the next time the transformation is run. All the traces that have references to the file after the block will also be generated in accordance with their new position in the file. The block comment tag (here '//') is controlled by environment settings and can be changed to match the target language.

The next sections elaborate on how this traceability information can be utilized and describe several scenarios.

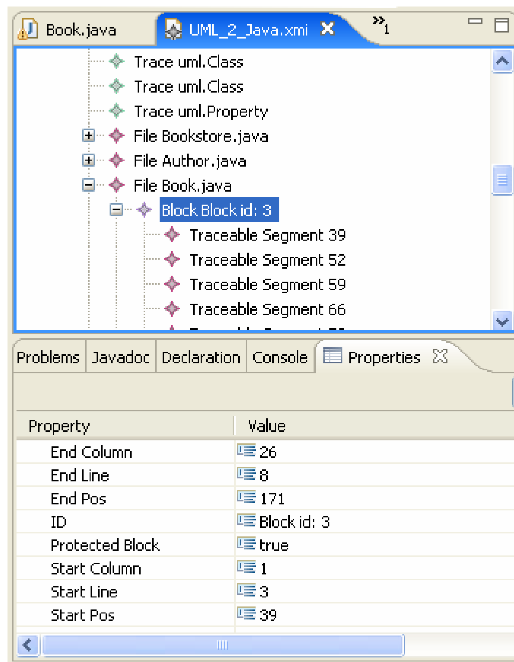

Fig. 3. Trace Model and Property View

3.3 Model to Text Specific Trace Scenarios

Trace links created in MOFScript between model elements and generated text can be utilized in several ways. This functionality can be used by different stakeholders, such as a *Transformation Architect* or *System Engineer*.

Extended Trace Inspection. The traceability model contains logical links from model elements to the code. These can be used to navigate and visualise traces in the code. For example, the user might select a specific model element and visualise the traces as highlighted code parts.

Coverage Analysis. Coverage analysis is useful for checking and ensuring that all relevant parts of the model are actually utilised by a transformation. If there are no traces from a particular model element, it is not used in the text transformation.

Impact Analysis. Impact analysis in text transformation can allow for checking the impact of a model change to existing generated code. A limitation in this regard is unprotected areas in the code that use model references, which cannot be seen from the traceability model.

Orphan Analysis. Orphans can occur in the code if model elements are deleted. There will then be traces from old model elements to the code. The transformation needs to be re-run in order to synchronise the model, the code, and the traces.

Trace Documentation. The traceability model can be used to generate different kinds of traceability documentation, for example by generated HTML documents. Such trace documentation can be provided by reusable model to text transformations [11] that have the trace model as source.

Unprotected Block Checking. When an unprotected region in the generated code has been implemented, the corresponding block in the trace model should be updated to show that it is completed. This will enable the Project Manager to check for bottle-necks, presenting a view of the remaining unprotected blocks that needs to be implemented, and if necessary move resources to a different part of the project.

Merging Traceability Models. Merging of traceability models may be used when several different transformations are executed from the same source model. The traces reference the same model elements, but sets of different target files. A merging of these will provide a more complete view of the traces from that particular source model.

Traceability Model Evolution. As models evolve, so will traceability models generated from those models. Histories of traceability models associated with a model may be used to analyse the evolution of the model with respect to code generation.

Our aim is to provide a toolset that supports the identified scenarios. Currently, we have developed a prototype that addresses some of the scenarios.

3.4 Traceability Analysis Prototype

The Traceability Model Analysis prototype is an initial version of a more complete traceability tool that also will consist of a repository for storing trace models. At this time, only MOFScript-specific trace analysis is supported. The prototype makes it possible to browse the source model in a tree editor and invoke different functionality on selected elements (Fig. 4).

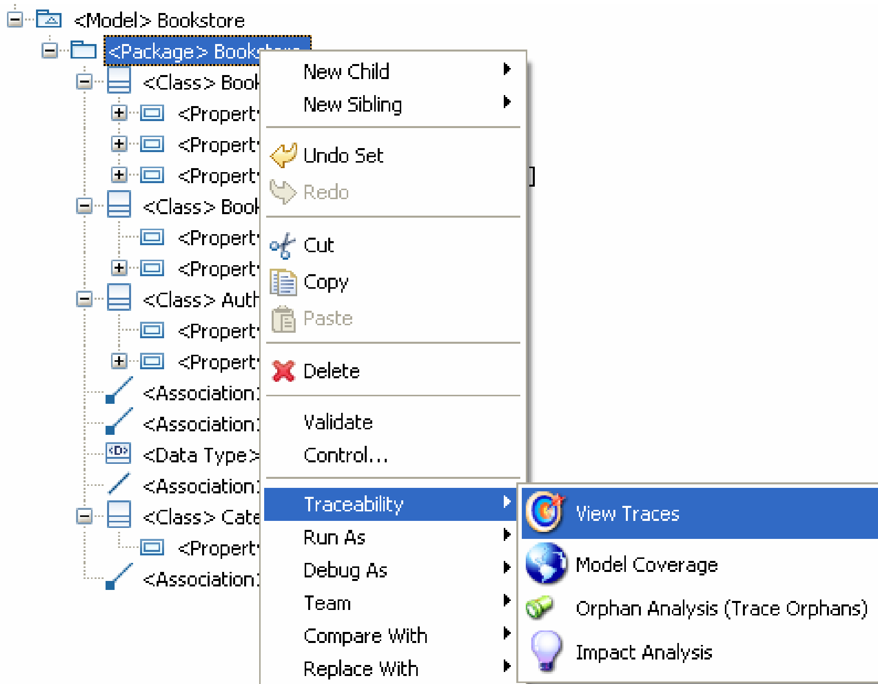

Fig. 4. Trace Menu for Model Bookstore

View Traces. This functionality gives a view of the traces for a selected model element (and its descendants). It can be used for trace inspection to locate the traces

View Traces Element: Package Bookstore

Traces	File	Block
Class:Book		
Class:Author		
trace(name): 5:14-20	Author.java	Block id: 2
Property:authorOf		
Property:name		
trace(type): 7:13-19	Author.java	Block id: 2
trace(name): 7:21-25	Author.java	Block id: 2
Class:Category		
Class:Bookstore		
trace(name): 5:14-23	Bookstore.java	Block id: 0
Property:books		
trace(type.name): 8:31-35	Bookstore.java	Block id: 0
trace(name): 8:37-42	Bookstore.java	Block id: 0
Property:categories		
trace(type.name): 9:31-39	Bookstore.java	Block id: 0
trace(name): 9:41-51	Bookstore.java	Block id: 0
Property:authors		
trace(type.name): 7:31-37	Bookstore.java	Block id: 0
trace(name): 7:39-46	Bookstore.java	Block id: 0

Fig. 5. Trace View

for specific model elements. Fig. 5 shows an example of this view for the Bookstore example, showing all traces generated from elements contained in the Bookstore package. As can be seen, the Bookstore class has several trace links to code segments in the file Bookstore.java.

Model Coverage. This functionality shows which parts of the model that do not have trace representations in the traceability model. It allows for checking that all intended model elements have been processed by the transformation. Fig. 6 shows the result of a Model Coverage analysis after adding two properties to the Book Class (isbn and price).

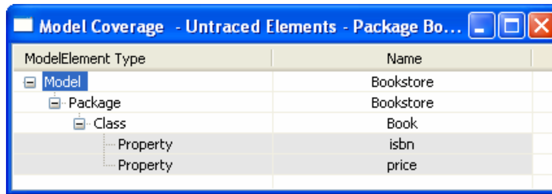

Fig. 6. Model Coverage

Orphan Analysis. This functionality allows for checking traces that are no longer valid, in that they reference model elements that no longer exist. Fig. 7 shows the result of an Orphan Analysis after the property *name* has been removed from the model element *Author*. Following the MDD approach, the normal procedure would be to rerun the transformation. However, on trace links that are created manually, this will be a useful feature.

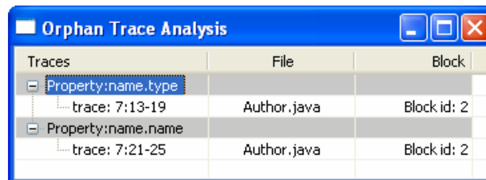

Fig. 7. Orphan Analysis after deleting a property

Impact Analysis. The functionality provided by the impact analysis checks for references in the generated code that will be affected by a modification of the model. This is basically an application of the *view traces* functionality with the source element as input to the query.

Functionality on Generated Files. The generated files also have traceability-specific actions that can be performed; this includes *view traces to file*, which will display the traces and the source elements that have this file as target and *view unprotected blocks,* which provides the user with a presentation of the unprotected blocks in the given file. The displayed unprotected blocks can be used for direct navigation into the file on the unprotected block's starting position. When the number of files and

unprotected regions are many (e.g., in complex systems), this functionality will simplify the manual development task.

When text has been inserted into an unprotected block manually, the traceability model can be updated to reflect the new positions of the traceable blocks and segments in the generated file.

4 Related Work

Even though traceability is a well known problem in software engineering and the current literature contains ample publications describing the need for traceability solutions, little work has been done in the field of model to text traceability. The OMG MOF Model to Text Transformation Specification [5] specifies a trace solution with the use of trace blocks, but currently there are no implementations of the standard available. How these trace links can be utilized is not described.

Acceleo Pro Traceability [12] is a traceability tool developed by Obeo that handles traceability links between model elements and code and vice versa. This tool enables round trip support; updates in the model or the code are reflected in the connected artefacts. Analyses are also available using the traces as input, but since this is a commercial tool, restricted information describing the solution is available. It seems to be based on similar ideas as described in [10] where model elements are traced to exact positions in files.

In [13] Alexander Egyed describes a bottom up approach for trace link generation with the use of the Trace Analyser tool. The approach requires the existence of a system that is both observable and executable, a list of artefacts from the development (e.g., model elements), usage scenarios or test cases and some initial traces that links the artefacts and the scenarios. This solution creates traces from lines of executed code to requirements and thus enables traceability among all artefacts.

Reqtify [14] is a requirement traceability tool from ChiasTek. It supports traceability through the entire project from high-level requirements to models, code, test scripts, and test results.

Objecteering 6.0 from Softeam [15] provides trace functionality through a trace editor that enables the user to create traces manually between artefacts. Model elements created from wizards based on existing elements can be traced automatically.

Rational RequisitePro [16] is a requirement and use case management tool that provides the ability to display traces between parent/child relationships and showing requirements that may be affected by upstream or downstream changes.

CaliberRM from Borland [17] is also a requirement management tool that enables manual creation of trace links from top level requirements to lower level descriptions.

5 Conclusion and Future Work

This paper presents a traceability solution for model to text transformations. Usage scenarios show that the solution is viable and how the generated trace model can be utilized.

The trace generation in MOFScript is implicit, meaning that all references to model elements are traced. In the MOF Models to Text Specification [5] the creation of traces is done explicit thru the use of trace blocks (it does not state that implicit traces

can not be used in addition). We believe that each approach has pros and cons and that an optimal solution should support a combination of both. Direct references from model elements to text should be generated automatically with the granularity defined by environment settings. There may also be situations where this is not sufficient to produce all trace dependencies required; therefore, explicit creation of trace links should be supported and classified accordingly. This functionality will be supported in a future version of the toolset.

In [18], Antoniol et al. have identified several challenges that must be addressed related to different aspects of traceability in MDD. Keeping trace information up to date can be an inconceivable task that often makes the links erode into an inaccurate state. The granularity of the trace links is also identified to be a challenge. The more fine-grained the trace links are, the more error prone they become. However, when traces are automatically generated, they are updated when the model changes and the code is regenerated. In this work, the challenge of keeping the trace links updated is addressed and the granularity issue is reduced.

The quantity of traces may be a challenge. In our solution, we are tracing all model element references and the number of traces might become incomprehensible and hence less useful. Furthermore, it might be a performance issue when the models and the transformation become large and complex. By adding a filtering mechanism to the traceability engine, it is possible to specify kinds of model elements that are interesting to trace and minimise performance overhead and unnecessary trace information.

Classification of traces was discussed earlier. In many traceability scenarios it may be useful or even essential to have meta information associated with traces, but it depends on the usage context. In an end-to-end traceability scenario involving different tools and artefacts, meta information will be important in order to distinguish the different traces. The scenarios we have shown here demonstrate usefulness of traces without classification.

The presented traceability solution implemented in MOFScript has its own specific traceability metamodel as shown in Fig.2. Future work includes the specification of a more generic trace metamodel (not specific to model to text) that will be implemented in a traceability tool. This tool will provide a simple interface for trace establishment both manually by users and automatically from several different MDD tools. The first step will be to integrate the trace establishment from MOFScript and then provide a user friendly interface for manual trace establishment.

Furthermore, we will investigate how to support trace model merging. Tools that do not support the provided interface of the traceability tool can supply the populated trace model, and the model can be merged into the repository's model representation. The goal will be to establish a MDD tool chain that in turn will populate the same project trace model. A typical scenario will be to model use-cases and have textual descriptions in Word documents, these artefacts will be traced to each other by manual establishment of the links. The use-cases will then be refined to different models and the proper trace links will be created manually or automatically by tools. The new models become sources to model to model transformations (e.g., an ATL transformation), which can populate the trace model in the repository. The generated target models will then act as source models to a transformation in MOFScript. The MOFScript transformation creates new trace links from the already existing model artefacts in the trace model to files, blocks and traceable segments.

With this approach, the development chain is capable of supporting end-to-end traceability where most traces are automatically created. Several end-to-end analyses similar to the model to text specific will also be supported.

Acknowledgements. This work is a result from the MODELPLEX project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme. (http://www.modelplex-ist.org/). Information included in this document reflects only the authors' views. The European Community is not liable for any use that may be made of the information contained herein.

References

1. Oldevik, J., et al.: Toward Standarised Model to Text Transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, Springer, Heidelberg (2005)
2. IEEE, IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990. 78 (1990)
3. Egyed, A.: Resolving Uncertainties during Trace Analysis. 12th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 3–12 (2004)
4. Jouault, F.: Loosely Coupled Traceability for ATL. ECMDA 05 Traceability Workshop (2005)
5. OMG, MOF Models to Text Transformation Language Final Adopted Specification Member doc: 06-11-01 (2006) www.omg.org
6. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On-Demand Merging of Traceability Links with Models. ECMDA 06 Traceability Workshop Bilbao (2006)
7. Walderhaug, S., et al.: Traceability Metamodel and System Solution. ECMDA 06 Traceability Workshop Bilbao (2006)
8. Aizenbud-Reshef, N., et al.: Model traceability. IBM Systems Journal 45(3), pp. 515–526 (2006)
9. Ramesh, B., Jarke, M.: Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering 27(1), pp. 58–93 (2001)
10. Oldevik, J., Neple, T.: Traceability in Model to Text Transformations ECMDA 06 Traceability Workshop Bilbao (2006)
11. Olsen, G.K., Aagedal, J., Oldevik, J.: Aspects of Reusable Model Transformations. ECMDA 06 Workshop on Composition of Model Transformations (2006)
12. OBEO, Acceleo Pro Traceability (2007), http://www.acceleo.org/pages/additionnal-products/en
13. Egyed, A.: A Scenario-Driven Approach to Trace Dependency Analysis. IEEE Transactions on Software Engineering 29, 17 (2003)
14. Chiastek, Reqtify (2007), http://www.chiastek.com/products/reqtify.html
15. Softeam, Objecteering 6.0 Web-Page (2007), http://www.objecteering.com/objecteering6.php
16. Software, I.R., Rational RequisitePro: reqpro/ (2007), http://www-306.ibm.com/software/awdtools/
17. Borland, CaliberRM (2007), http://www.borland.com
18. Antoniol, G., et al.: Problem Statement and Grand Challenges in Traceability. Center of Excellence for Traceability (2006)