

# Towards Rigorously Defined Model-to-Model Traceability

Nicholas Drivalos<sup>1,2</sup>, Richard F. Paige<sup>1</sup>, Kiran J. Fernandes<sup>2</sup>,  
Dimitrios S. Kolovos<sup>1</sup>

<sup>1</sup> Department of Computer Science, University of York  
{nikos, paige, dkolovos}@cs.york.ac.uk

<sup>2</sup> The York Management School, University of York  
kf501@york.ac.uk

**Abstract.** A Model Driven Engineering process typically involves models expressed in different modelling languages that capture different views of the system under development. To enhance automation, consistency and coherency, establishing and maintaining semantically rich traceability links between model elements that belong to different models used throughout the process is of paramount importance. In this paper we propose a rigorous approach to defining such semantically rich traceability links between models expressed in diverse modelling languages using case-specific traceability metamodels and demonstrate the practicality and usefulness of our approach using a concrete example.

## 1 Introduction

Model-Driven Engineering (MDE) is an approach to software development where the primary focus is on models, as opposed to source code. A model describes certain views of the software system and its environment at a certain level of abstraction. MDE uses models to represent all artefacts that are involved in the software development process, such as requirements, software components or system documentation. Usually, models are described by different languages and they can be refined, evolved into a new version, and used to produce other models or even executable code. The ultimate goal of MDE is to raise the level of abstraction, and to develop and evolve complex software systems by manipulating models [9].

In a typical MDE process, many different and heterogeneous modelling artefacts are produced. This poses challenges to the traceability of the various models elements, i.e. the ability to establish, represent and update relationships among the various artefacts developed during the software development life-cycle. Traceability is considered as a necessary system characteristic since it underpins software management, software evolution and validation [14].

In this paper we present an approach to specifying and capturing strongly typed and semantically rich traceability links between models that conform to potentially different metamodels. In our approach we use a dedicated traceability

metamodel to specify type safe traceability links of interest, as well as a set of inter-model constraints for verifying the validity of the established links.

The rest of the paper is organised as follows: In Section 2, we define the main concepts, which are met throughout this paper, while in section 3 we present the two approaches to storing and managing traceability information, as well as the advantages and disadvantages of each approach. In Section 4, we propose the main contribution of this paper, which is the use of case-specific traceability metamodels and inter-model constraints, that define strongly typed and semantically rich traceability links. In Section 5, we present a concrete example that demonstrates the practicality and usefulness of our approach, while in section 6 we provide a discussion on related work and compare to our approach. Finally, in Section 7 we conclude and identify interesting further work on the subject.

## 2 Background

Traceability is defined in the IEEE Standard Glossary of Software Engineering Terminology [16] as follows:

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.

The above definition exhibits a strong influence from the originators of the traceability concept, the requirements management community. However, for the purpose of traceability in the context of MDE a broader definition is required. Such a definition is provided in [2], where traceability is considered as any relationship that exists between artefacts of the software development life-cycle, such as mappings generated as a result of transformations or relationships that are computed based on existing information (e.g. dependency analysis).

Following [10], there are two strategies to storing and managing model-to-model traceability information. In the first one, the traceability information is embedded in the models they refer to, while in the second strategy the traceability information is stored separately from the models.

### 2.1 Intra-Model Storage of Traceability Links

Under this strategy, the traceability information is stored within the artefact they refer to in the form of model elements or model element attributes, such as tags and properties. Despite its simplicity and human friendliness, keeping such information with the artefacts can be problematic for several reasons. If the link is directed and stored in the source model only, it is not visible from the target model. On the other hand, if the traceability information is stored in both models, then this information must be maintained consistent, thus the burden

of maintaining consistency is introduced every time a change occurs [2]. In addition, embedding traceability information inside a model causes “*pollution*” [10], i.e. the inclusion of elements in the model of secondary importance. Such an inclusion can render a model overcrowded and can make it difficult to understand and maintain. Finally, the issue of *uniformity* arises in this approach [1]. In an MDE environment, it is common that models have their own representations and semantics. Hence, it is very difficult to distinguish the traceability information from the other model artefacts. As a result, automated analysis of traceability information becomes very challenging. The main approaches falling under this strategy utilise mainly language specific constructs. For example, specific types of traceability links are represented in UML diagrams by using stereotyped dependencies, such as  $\ll \textit{refines} \gg$  [8].

## 2.2 External Storage of Traceability Links

Under this strategy, traceability information is stored separately from the models they refer to in a separate model. Constructing such models has two clear advantages. First, the source and target models remain “clean”, since the traceability links are stored in a separate model, whose concern is to capture this kind of information. In this way, the aforementioned “pollution” is avoided. In addition, storing traceability links in a model who conforms to a metamodel with clearly defined semantics makes automatic analysis by tools much easier. A prerequisite for storing traceability links externally from the models they refer to, is that the various model elements have unique identifiers, so that the related traceability links can be resolved unambiguously [10]. For example, such a mechanism is provided by MetaObject Facility (MOF) [15] and Eclipse Modeling Framework (EMF) [6] in the form of a *xmi.id* identifier. In our view, storing traceability links in separate models is more preferable than embedding the traceability information in the models they refer to. This is because in addition to the aforementioned advantages, this strategy is able to capture both intra-model as well as inter-model links.

## 3 A Generic Approach to Inter-Model Traceability Establishment

In the general case we need to establish links between elements belonging to a number of models that potentially conform to diverse metamodels. Also, several types of traceability links linking different types of model elements may need to be captured. As discussed, traceability links should not be captured by constructs internal to the models they refer to, to reduce “pollution”. Instead, they should be captured in the form of a separate model. This model that contains the traceability links must conform to a metamodel. There are two alternatives: use a general-purpose traceability metamodel or use multiple case-specific traceability metamodels.

### 3.1 General-Purpose Traceability Metamodel

In this case, a generic metamodel that enables capturing relationships between any types of model elements is used. In this metamodel, a traceability link can connect any number of elements, of any type in any model. Such a metamodel is the *Unified Traceability Scheme* developed in [12]. The main advantages of a general-purpose metamodel are simplicity and uniformity (as all traceability models conform to the same metamodel) which creates potentials for enhanced tool-interoperability as tools will be able to import, export and manage traceability in a common format. On the other hand, such a general purpose metamodel does not capture case-specific strongly typed traceability links with rigorously defined semantics and constraints, and thus allows establishment of potentially illegitimate traceability links. For example, in the case we want to represent traceability links between a class diagram and a relational database model and we know that links exist between classes of the former model and tables of the later, a generic traceability metamodel allows the establishment of possibly illegitimate links, such as class-column links. Provision of extension mechanisms along with the general-purpose traceability metamodel is an approach often used to allow better support for case-specific requirements. Examples of this approach can be efficient in tackling the aforementioned issues [3, 13]. However, they still lack the efficiency of case-specific metamodels to capturing case specific information and semantics.

### 3.2 Case-Specific Traceability Metamodel

In this case, for each traceability scenario, a case-specific traceability metamodel is defined. This traceability metamodel captures case-specific strongly typed traceability links with well-defined semantics that potentially include correctness constraints that extend beyond simple type conformance. Apparently, such a metamodel can capture more case-specific information and semantics and, due to its strongly typed nature and the attached constraints, restricts users and tools to establishing legitimate traceability links only. By contrast, a case-specific traceability metamodel requires some effort to be spent for its construction and also tools that support different traceability metamodels shall not be able to directly communicate with each other. Nevertheless, the process of establishing an explicit traceability metamodel is to our view beneficial in the long term as it involves the engineers in a cognitive process through which the conceptual correspondences between the respective metamodels that are involved in the traceability scenario are enhanced. Also, with the advent of model management technologies (e.g. model transformation languages) transforming between different well-defined traceability metamodels is expected to be a straightforward process to automate.

## 4 Establishing a Strongly Typed, Semantically Rich Traceability Metamodel

In this paper we propose an approach for capturing and representing strongly typed and semantically rich traceability links. This approach involves establishing a traceability metamodel that defines strongly typed traceability links and a set of constraints that express validity requirements that can not be captured by the metamodel itself.

To be strongly typed, the traceability metamodel needs to explicitly refer to types of elements defined in other metamodels. For example, consider that we need to define a traceability metamodel that enables establishment of simple traceability links between instances of A (from MMA) and instances of B (from MMB), but no links between two instances of A or two instances of B. To capture such a metamodel, the underlying modelling technology must not consider each metamodel as a closed space - but instead allow inter-metamodel references. An exemplar technology that supports inter-metamodel references is the Eclipse Modeling Framework (EMF) [6].

Although a metamodel captured using a modelling technology that allows inter-metamodel references can enforce type safety, there are often additional requirements that need to be specified, and which the traceability metamodel cannot capture by itself. For instance, in the previous example, an additional requirement may be that each instance of A from MMA can only be linked to no more than one instances of B in MMB. To specify such constraints, a constraint specification language that can express constraints spanning over elements belonging to a number of models of potentially different metamodels is required. The Object Constraint Language (OCL) [17] currently lacks such capabilities as it does not provide constructs for expressing cross-model constraints. Exemplar constraint languages that support establishing cross-model constraints include the Epsilon Validation Language (EVL) [11] and the XLinkit [4] toolkit.

The combination of a strongly typed traceability metamodel with verifiable inter-model constraints restricts users and tools to establishing and maintaining only meaningful traceability links, which can be automatically validated to discover potential omissions and inconsistencies. Such issues can arise either during the establishment of the traceability links or later on in the lifecycle of the models among which traceability links have been established. In the next section, we demonstrate the practicality and usefulness of our approach using a concrete example.

## 5 Example

In this section, we present the proposed methodology using a concrete example. In our example, we use two simple metamodels, the *ClassMetamodel* and the *ComponentMetamodel*, which are illustrated at the top and bottom of Figure 1 respectively. Our aim is to capture traceability links between instances of Package from the *ClassMetamodel* and Component from the *Component metamodel*,

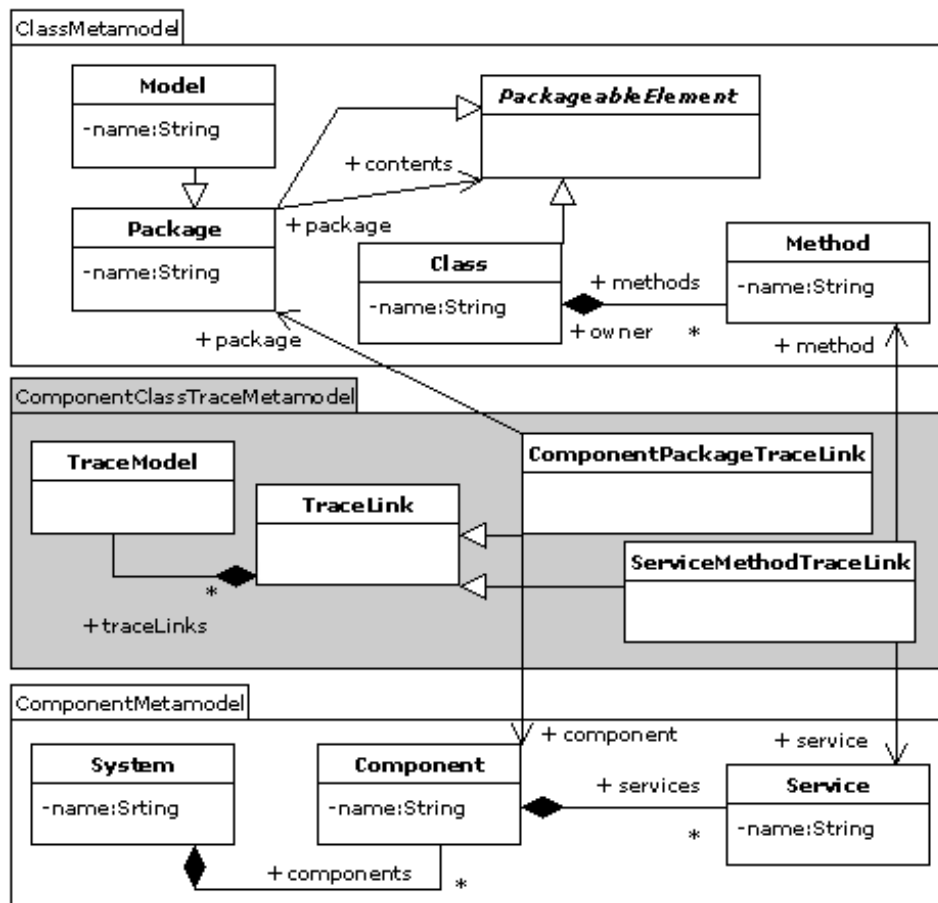


Fig. 1. The ClassMetamodel, ComponentMetamodel and ClassComponent-TraceLinkMetamodel metamodels

and links between instances of Method from the *ClassMetamodel* and Service from the *ComponentMetamodel*. Furthermore, the following exemplar constraints must be satisfied by the trace model (*ComponentClassTraceModel*):

- (C1) For each Service in *ComponentModel* there is exactly one *ServiceMethodTraceLinkTrace* in the *ComponentClassTraceModel* that links it with a Method in the *ClassModel*
- (C2) If a Service in the *ComponentModel* is linked to a Method in the *ClassModel* via a *ServiceMethodTraceLink*, then the component in which the service is defined must also be linked with the *Package* in which the *Class* that contains the Method is defined.

The first step of the solution is to define the *ClassComponentTraceMetamodel* case-specific traceability metamodel that is displayed as a shaded pack-

age in the middle of Figure 1. The metamodel specifies a *TraceModel* container, an abstract *TraceLink* class, and the *ComponentPackageTraceLink* and *ServiceMethodTraceLink* classes that extend *TraceLink*. The *ComponentPackageTraceLink* defines two references: the package reference that is of type *Package* and the component reference which is of type *Component*, from the respective metamodels. Similarly, the *ServiceMethodTraceLink* defines the service and method references which are of type *Service* and *Method* respectively. By specifying explicitly the supported traceability links, we preempt establishment of illegitimate links (e.g. tracing a *Component* to a *Method*).

However, the metamodel cannot enforce the additional constraints C1, C2 discussed above by itself. To impose such constraints, we use the Epsilon Validation Language [11], a constraint language that is capable of expressing constraints over multiple models of different metamodels simultaneously.

In Listing 1.1 constraint C1 applies to all instances of *Service* in the *ComponentModel* and in line 5 it calculates the number of *ServiceMethodTraceLink* in the *ComponentClassTraceModel* that link the *Service* with a *Method* from the *ClassModel*. In line 6 it returns true if exactly one trace link is found and false otherwise. Then in line 8 it reuses the *count* variable calculated in the *check* part of the constraint to generate an appropriate error message according to whether zero or more than one trace links have been found.

**Listing 1.1.** Constraint C1 expressed in EVL

```

1 context ComponentModel!Service {
2   constraint C1 {
3     check {
4       var count := ComponentClassTraceModel!ServiceMethodTraceLink.
5         all.select(sml|sml.method = self).size();
6       return count = 1;
7     }
8     message {
9       if (count = 0) {
10        return 'Service ' + self.name + ' does not trace to a method';
11      }
12      else {
13        return 'Service ' + self.name + ' traces to many methods';
14      }
15    }
16  }

```

In Listing 1.2 constraint C2 is evaluated against all instances of *ServiceMethodTraceLink* in the *ComponentClassTraceModel* and checks that there exists at least one *ComponentPackageTraceLink* that links the component in which its *service* is defined with the *Package* in which the class that contains the *method* is defined.

**Listing 1.2.** Constraint C2 expressed in EVL

```

1 context ComponentClassTraceModel!ServiceMethodTraceLink {
2   constraint C2 {

```

```

3     check : ComponentClassTraceModel!ComponentPackageTraceLink.all
4         .exists(cpl|self.service.component = cpl.component
5             and self.method.owner.container = cpl.package)
6     message : 'The package in which method '
7         + self.method.name + ' is defined does not trace '
8         + 'to the component in which service '
9         + self.service.name + ' is defined'
10 }
11 }

```

In this example we have demonstrated our approach on a simple but representative example. To establish rigorous traceability links between elements of the exemplar *ComponentMetamodel* and *ClassMetamodel* metamodels we have introduced a new metamodel (*ComponentClassTraceMetamodel*) in which every legitimate type of traceability link is represented as a separate metaclass that contains references to the types of elements it can link. Moreover, we have demonstrated that additional constraints which cannot be captured by the traceability metamodel are necessary for rigorously specifying legitimate traceability links, and shown how such constraints can be captured using a constraint language (EVL) that supports accessing multiple models simultaneously.

## 6 Related Work

### 6.1 Atlas Model Weaver

AMW, the Atlas Model Weaver [13], is a tool created by INRIA as part of the ATLAS Model Management Architecture . Its primary goal is to capture and to store links between models. This information is stored in a model, which is called *weaving model*. This metamodel is very flexible and may be extended to add additional mapping semantics. The process of creating the weaving model can be manual or semi-automatic. AMW provides a base weaving metamodel enabling to create links between model elements and associations between links. The main difference of AMW and the approach proposed in this paper is the fact that AMW treats the weaving model as a closed space and does not allow inter-metamodel references, while our approach focuses on those references since they provide the basis for a more semantically rich weaving metamodel.

### 6.2 C-SAW

The Constraint-Specification Aspect Weaver (C-SAW) is a general transformation engine for manipulating models and is a plug-in for GME [19]. This approach is strongly influenced by the Aspect Oriented Software Development community. The main goal of C-SAW is to maintain consistency of complex evolving models. C-SAW offers the ability to explore numerous modelling scenarios by considering crosscutting modelling concerns as aspects that can be rapidly inserted and removed from a model. This permits a modeller to make changes more easily to the base model without manually visiting multiple locations in the model.

Comparing C-SAW to our approach, there are two differences. C-SAW focuses on specifying *side-effects* rules to deal with model changes, while our approach is focused on capturing various links among different model elements. In addition, C-SAW does not consider explicitly inter-metamodel references.

### 6.3 XWeave

XWeave is a model weaver based on EMF's ECore meta metamodel [7]. XWeave takes a base model as well as one or more aspect models as input and weaves the content of the aspect models into the base model. XWeave finds pointcuts either by name matching or by defining them with the oAW expression language. Similarly to C-SAW, XWeave uses an automated approach to identifying links between elements of different models while our approach is focused on manual, rigorous trace link establishment.

## 7 Conclusions & Future Work

In this paper we have presented an approach to establishing trace links between models expressed using different modelling languages. We have proposed using case-specific traceability metamodels that define strongly typed and semantically rich traceability links and inter-model constraints to ensure the soundness of the established links.

We have used the proposed technique for defining rigorous traceability links between requirements specifications expressed using models conforming to the non-trivial  $i^*$  [18] and KAOS [5] metamodels. Throughout this work we have identified a number of interesting reoccurring patterns in case-specific traceability metamodels. In the future we plan to encapsulate these patterns in a dedicated language that enables specifying traceability metamodels with rigorously-defined semantics at a higher level of abstraction.

## References

1. Future Research Topics Discussion. Traceability Workshop, EC-MDA, November 2005. <http://www.sintef.no/upload/10558/Future-Research-Topics.pdf>.
2. N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafini. Model Traceability. *IBM Systems Journal*, 2006.
3. B Vanhooff, S Van Baelen, W Joosen, Y Berbers. Traceability as Input for Model Transformations. In *Proc. Traceability Workshop, European Conference in Model Driven Architecture (EC-MDA)*, 2007.
4. Christian Nentwich, Licia Capra, Wolfgang Emmerich and Anthony Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, May 2002.
5. R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. *Proc. ICSE'98 - 20th International Conference on Software Engineering, Kyoto*, 1998.

6. Eclipse Foundation. Eclipse Modelling Framework Project (emf). <http://www.eclipse.org/modeling/emf>.
7. I. Groher and M. Voelter. Models and Aspects in Concert. In *Proc. of the 10th Workshop on Aspect-Oriented Modeling co-located with the 6th International Conference on Aspect-Oriented Software Development (AOSD'07)*, Vancouver, Canada, ACM Press, 2007.
8. W. Heaven and A. Finkelstein. A UML Profile to Support Requirements Engineering with KAOS. *IEEE Proceedings: Software*, 2004.
9. S. Kent. Model Driven Engineering. In *Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, Proc. of Third International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of LNCS, pages , Springer., 2002.
10. D. S. Kolovos, R. F. Paige, and F. Polack. On Demand Merging of Traceability Links with Models. In *ECMDA - TW: Traceability Workshop, at European Conference on Model Driven Architecture, Bilbao, Spain*, 2006.
11. D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Eclipse Development Tools for Epsilon. *Eclipse Summit Europe, Eclipse Modeling Symposium*, 20062.
12. A. E. Limon and J. Garbajosa. The Need for a Unifying Traceability Scheme. In *Proc. Traceability Workshop, European Conference in Model Driven Architecture (EC-MDA)*, 2005.
13. D. Del Fabro Marcos, J. Bézivin, F. Jouault, E. Breton, E., and G. Gueltas G. AMW: a Generic Model Weaver. In *Proc. of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles*, 2005.
14. L. Naslavsky, T. A. Alspaugh, D. J. Richardson, and H. Ziv. Using Scenarios to Support Traceability. In *TEFSE '05: Proc. of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering, New York, NY, USA, 2005*, ACM Press, 2005.
15. OMG. Metaobject Facility. <http://www.omg.org/mof>.
16. IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. IEEE , New York, 1990.
17. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.
18. E. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. *Proc. RE-97 - 3rd International Symposium on Requirements Engineering, Annapolis*, 1997.
19. J. Zhang, Y. Lin, and J. Gray. Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine. In *Model-Driven Software Development - Research and Practice in Software Engineering. Springer Verlag*, 2005.