RT-MDD Framework – A Practical Approach

Marco Costa[1], Alberto Rodrigues da Silva[2]

[1] Univ. Lusíada
Rua da Junqueira, 188-198
1349-001 Lisboa, Portugal
mbcc@acm.org

[2] INESC-ID/Instituto Superior Técnico
Rua Alves Redol, Nº 9
1000-029 Lisboa, Portugal
alberto.silva@acm.org

Abstract. Traceability in software engineering has been handled mostly has a documentation activity. This paper describes a reactive approach to traceability using MDD as general orientation and QVT as a transformation language. A conceptual model is presented defining some of the key concepts of a reactive traceability framework. Also the semantics of these key concepts is explained. Some examples are given of the framework's artefacts.

Keywords: Reactive traceability, MDD, QVT, UML, RT-MDD

1 Introduction

An information system involves a set of active artefacts that cooperate towards a common goal. These artefacts are present in multiple views, regarding the abstract layer we take as a viewpoint. For instance, a system may be described by its functions, data structures and technologies, among other features. Documentation is a part of the solution, just as the formulation of a problem may be considered part of its solution.

Nowadays information systems are growing in complexity, i.e., in terms of the number of artefacts and relations between them. Development of new applications and maintenance of the existing ones should be accompanied by tools and methodologies which minimize the risk of introducing a state of incoherence between the artefacts. When this problem is not tackled the reality shows that programming code evolves with no or minor relation with models. In large applications, when models are almost completely outdated the system is in risk of becoming unmaintainable.

Traceability, is used in many different contexts from food industry to software development and maintenance. We focused our work in traceability between artefacts of models and code (Fig. 1). The requirements traceability [11] was left outside the

solution for now, but this issue may be integrated in the future work, as other types of traceability.

Traceability deals with keeping records of relations between artefacts of the same, or different abstract levels. We propose the term *reactive traceability* as a characteristic of a system that not only keeps information of the relations, but also can react to and prevent changes to artefacts or its relations (traces). The goal of the RT-MDD (Reactive Traceability Model Driven Development) framework is to maintain a state of coherence between the artefacts of the system.

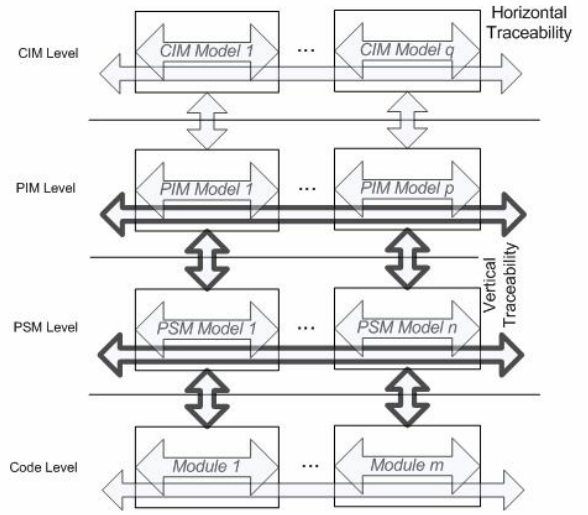

Fig. 1. Traceability and MDA (bolder arrows defines focus of this paper).

Automated transformation of models-to-models, models-to-code, as well as code-to-models, is becoming a reality [19, 20, 21]. These transformations generate output artefacts from source artefacts. The relation between input and output artefacts of a transformation is just one, and obvious, type of semantic relations among others. Object Management Group QVT (Queries, Views and Tranformations) [3] is aimed to standardize not only transformations with models but also other operations with models, like queries and views. Our approach takes QVT as a starting point to accomplish the construction of automated transformations between models and implements a way of maintaining traces between artefacts (models and code) as well as reacting to changes for the sake of system coherence.

Our proposal is aligned with OMG standards and recommendations like UML [5] and QVT [3]. Even if QVT by itself has not a complete implementation, it is however a starting point to different approaches and products like Tata ModelMorf [10] or Compuware OptimalJ [7]. Other concurrent approaches to QVT include ATL [8], Mistral [6] and EML [18].

Gorp's [12] *consistency* is a similar concept to coherence in this paper. The vision of traceability in a MDD perspective is shared with other initiatives, such as those in [14, 15, 16].

The paper is structured as follows. Section 2 proposes the conceptual model of a traceability solution and defines traceability from a systemic approach. Section 3 identifies different types of dependency relations between artefacts. Section 4 relates transformations with traceability and explains how QVT define traces. Section 5 describes some important features of the traceability solution prototype implemented.

2 Artefacts and Traceability

The relevant information for traceability (Fig. 2) corresponds to artefacts, which are products generated or crafted by tools used in a development process. In this context an artefact may be any piece of code, or model object, with relevant properties regarding traceability; for instance, an UML class, a C# method, or a SQL Server table definition.

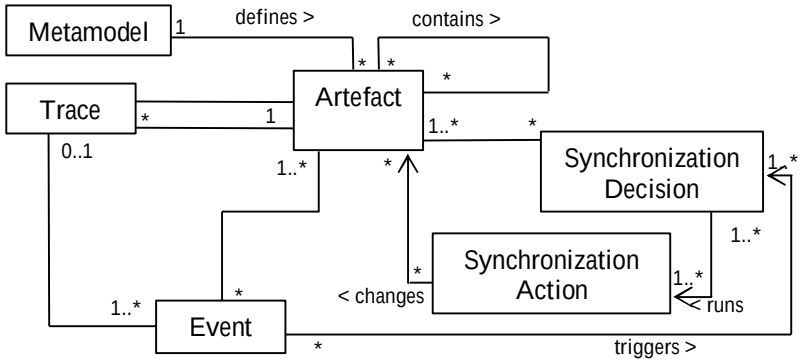

Fig. 2. Conceptual model of the reactive traceability framework.

The artefact can be any object used, modified or made in the context of the development environment, included in the system, and accessible by the reactive traceability solution. Artefacts should be defined through of a metamodel.

For a programming language to be included in this solution it must have a metamodel, even if not covering all possible concepts, if irrelevant to the level of traceability needed. Our approach uses MOF to express metamodels. Artefacts may be transformed in other artefacts, using some kind of transformation process available (fully automatic, assisted or manual) and traces are maintained when a transformation occurs. Of course, transformation is just one of different possible types of semantic relations between artefacts. Considering legacy systems, a transformation may not be possible if the code cannot be changed. In cases like this, maybe it is necessary to record manually different relations expressed (e,g, considering a table, named *client,* in the legacy system related to a UML class, named *entity* in a class diagram). As a consequence, there is a need of other types of dependency relations, other than transformations.

3 Dependency Relations and Traces

Different dependency relations are possible between artefacts. We can identify two types of classification: a) *conceptual level* and b) *semantic coupling*. In a) we can consider artefacts in different conceptual levels [4] (vertical traceability), in the same level (horizontal traceability) and within the same model or in different ones (intra and inter-model traceability) as seen in Fig. 1.

Each dependency relation can also be characterized by the level of coupling that exists between the artefacts. Let us define the *equivalence* relation between two artefacts of a model, or from different models, as when they have the *same representation* of the *same concept*. In this context *same representation* stands for having the same data that identifies the concept, even if the artefacts are in different levels of abstraction, or artefacts.

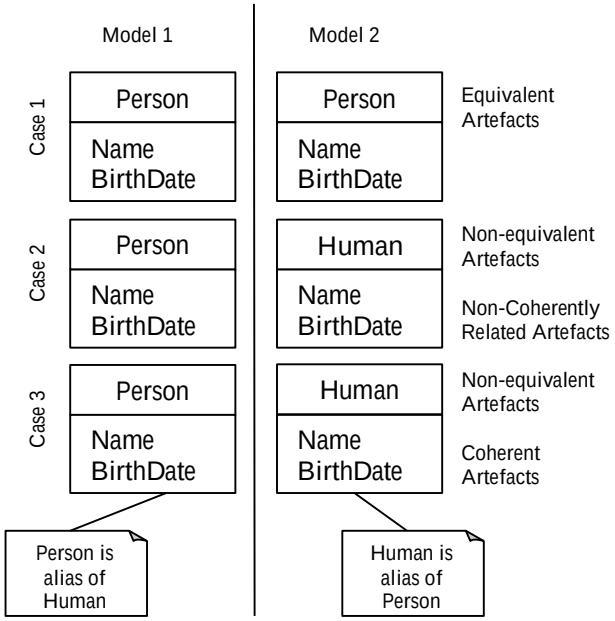

Fig. 3. Dependency relations between models.

Considering Fig. 3, there are three cases which illustrate differences between dependency relations. In *Case 1* the two artefacts are equivalent, as they have the same representation of the concept *Person,* considering different models. In *Case 2* the classes have different names so they are not equivalent. In the same case we can't say if they are coherent or not, as there is no information in the models relating the two classes. In *Case 3* the two concepts are coherent, with added simplified information to each class, as there is a semantic relation between the two artefacts. The semantic relation is not always trivial to observe, considering complex legacy applications, different natural language names to concepts, or even different

programming and design paradigms. In real work coherence is more difficult to maintain than the other two relations. It is however the kind of relation that is necessary to deal with if reactive traceability is to be maintained between the artefacts. Each change to an artefact may trigger an event which may change or not the coherence state of the system itself. If a part of a system is in an incoherent state, the system may be in risk. In consequence, the state of the system must be checked when a change of an artefact occurs and a decision may be required to run existing actions that will bring again the system to a coherent state.

In this perspective a trace is seen as a record of a dependency relation between two artefacts. The artefacts may have different levels of abstraction, e.g., it is possible to define a trace between all classes of an ULM class model and a set of table definitions. The artefacts can be at a metalevel, i.e., at a language definition level (e.g. artefact *Class* of metamodel *UML)*. As the artefacts also include structural artefacts (e.g. the UML class X) generic rules of traceability may not old for all necessary traces. A reactive traceability solution must maintain the traces and ensure that they are still valid at any time.

4 Reactive Traceability and Transformations

Transformations between models or between models and code (usually *from* models *to* code and not the opposite) are a relevant issue to reactive traceability. After transformations are performed traces are created (implicitly or explicitly) but this is not the only action that can create traces. There are two viewpoints (Fig. 4) to the issue of creating traces: a) in legacy applications one may consider artefacts already existent and traces will instantiate and describe some implicit or explicit semantic relations, or dependencies, between them; b) from a starting point in an artefact (diagram or code) it is necessary to create (or generate) another artefact, using some type of transformation.. These two viewpoints are both necessary in a solution that implements reactive traceability. The first item has a focus on creating and maintaining the semantic relations between existing artefacts and the word *existing* is a keyword to understand this viewpoint. At some time of the development process changes in the system will become evolutions of an existing state. For each change the system coherence is checked and decisions are made about the new achieved system state. The second gives more importance to the generation process [1, 2] and traces are relations between *old* and *new* (generated) artefacts. The second viewpoint is related to the IEEE traceability definition in [13]: *"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".* When a generation of a set of artefacts occurs it is necessary to record the new dependency relations that are created in the process. These two approaches are not only valid but necessary, in a development process and reactive traceability must include both.

Transformations between models, and by extension between models and code, may be realized using a standard like QVT [3]. The QVT specification uses trace classes to record traceability data between artefacts. QVT has a declarative language (Relations

Language – QVT-RL), an imperative language (Operational Mappings Language – QVT-OML) and a simplified core language (QVT-Core) in which the other more complex language constructs may be expressed.

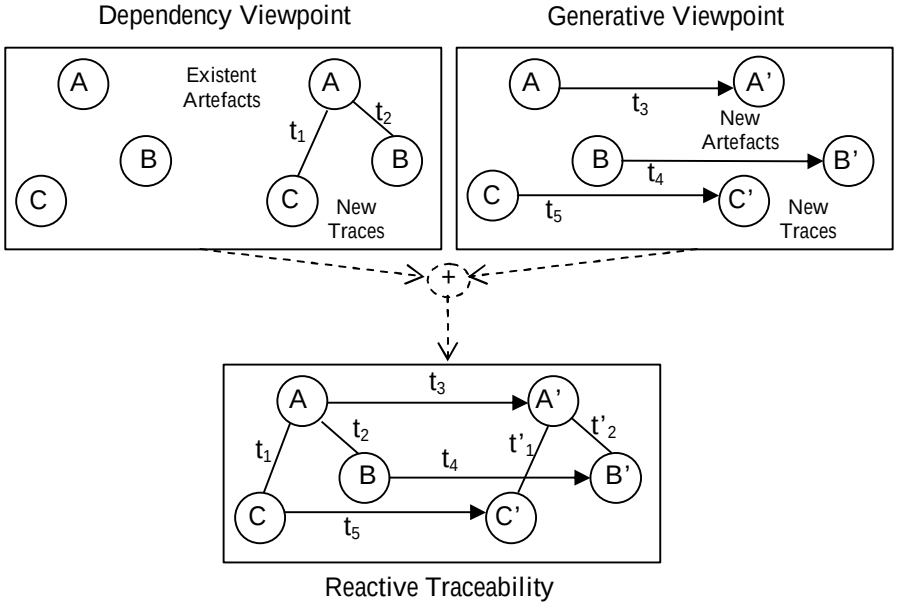

Fig. 4. Creation of traces: two approaches to reach reactive traceability.

In the QVT-RL the traces are implicitly defined (which, in this context, is almost the same as undefined), but there is a trace class generation rule that defines the corresponding trace class definition in the QVT-Core language. The generation rule states that [3] *"Corresponding to each relation there exists a trace class in core. The trace class contains a property corresponding to each object node in the pattern of each domain in the relation"*. Let us consider a simplified QVT relation between a class diagram at design level written in UML and the corresponding C# classes:

```
relation UMLClassToCSharpClass
{
   checkonly domain uml uc: UClass {
         namespace = un:UMLNamespace{},
         name= cn }
   checkonly domain csharp csc: CSClass {
         namespace = csn:CSNamespace{},
         name= cn }

}
```

The generated QVT-Core trace class is:

```
class TUMLClassToCSharpClass
```

```
{
   uc: UClass;
   un: UMLNamespace;
   csc: CSClass;
   csn: CSNamespace;
}
```

When instantiated, this trace class will represent the actual dependency relation between the two related artefacts.

5 RT-MDD Framework

As seen later, transformations are an important source of traceability information but are not the only one available. Different tools can create artefacts that may include other artefacts with traces that must be tracked and maintained. Considering transformations as a necessary tool, implemented by a *QVT Engine,* to efficiently produce models and code artefacts, Fig. 5 shows the high-level architecture of a reactive traceability solution. The RT-MDD framework has *providers* which are integration components to different tools. Of course, more components from different types may be added if they produce or consume artefacts with traces. The link between the Traceability Engine and each other component varies from type of provider. Each tool to be included in the traceability solution must have the necessary *provider plug-in* that will be used to access the relevant artefacts which have traces. The relations between artefacts are kept within the RT-MDD solution.

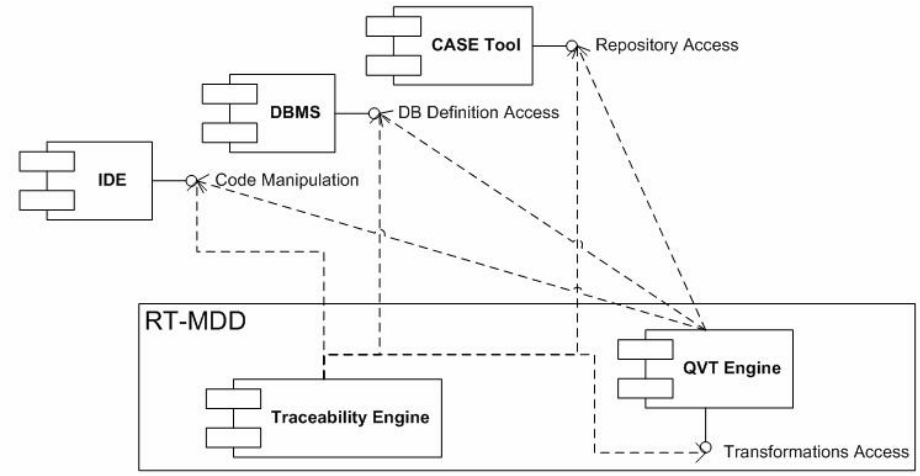

Fig. 5. UML component diagram of a traceability solution.

It is important to note, however that the traceability solution is not a production or test environment tool. It is designed to be used in development environments. As it

acts with the structure of the target applications, if the structure is not changed there is no need to control traceability.

The Traceability Engine (TE) permanently verifies the coherence state of the system in a three phase approach: (1) polling, (2) artefacts change handling, and (3) trace change handling.

(1) The TE polls each artefact. When a change is made to an artefact, the TE adds that artefact to a *list of changed artefacts.*

(2) For each artefact in the *list of changed artefacts* TE searches in an *artefact event list* all the relevant events related to that artefact. For each of these events found, it is necessary to verify if a valid action was made in the system. Possible valid actions are: *create*, *update* and *delete*. The *read* action is not valid as it does not change the system state. The *create* action is valid when used with metamodel artefacts. Only then it is possible to say if an event related directly to the artefact exists. If this is the case, the event is triggered. This phase ensures that actions over artefacts and their properties don't change the coherence state of the system.

(3) After artefact events are verified it is necessary to check if traces still hold (even after eventual artefact changes). For each artefact in the *list of changed artefacts* TE recursively searches trace events that refer artefacts contained in the artefact. If a trace event is found, and the relevant artefact and action hold, the event is triggered.

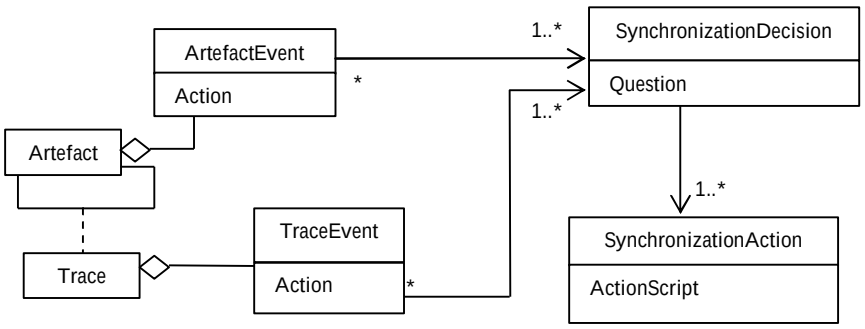

Fig. 6. Domain model of the trace engine

When the trigger conditions are satisfied TE runs synchronization decisions (Fig. 6). When the user selects one of the decisions available TE runs a set of synchronization actions associated to that decision. After the decision is taken the system is in equal or more coherent state than before. If a decision of solving a coherence issue is postponed, a warning is generated and logged in a *to-do list* for further processing.

The user interaction with the solution is minimized if trace and artefact events have only one synchronization decision (representing just one option). In that case TE may enforce or omit the related actions (in this case generating an entry in the *to-do list).*

A prototype is currently under development wich has a graphical user interface with QVT diagrams to define transformations and templates.

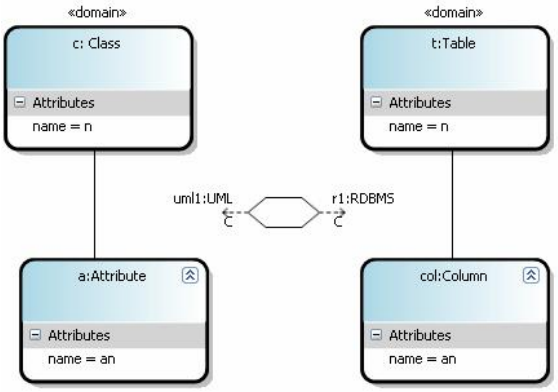

Fig. 7. A QVT Relations diagram (made with the prototype application)

6 Conclusions

Traceability is still regarded has a documentation activity. If the human resources involved in the project are not conscientious about the relevance of documentation in maintenance phases, traceability is not seen as a critic issue. In the software engineering area, traceability has to address specific issues, such has documentation, model production and maintenance, model and code transformations and artefact coherence. This work presents some issues that where considered in the production of RT-MDD framework.

From the early 1970s there is work on requirements traceability, usually with a predominant human operation. Nowadays, the presence of traceability can be very important for the acceptance of MDD and Model Driven Architecture (MDA) in an industry environment. Without tools supporting traceability, across all system levels, the traditional resistance [9, 17] in MDD acceptance will continue to exist.

In our perspective, traceability is not about documentation of the system, it should be more focused on the system itself, as well as its parts and the way they are related to each other. Reactive traceability is a proposal with that orientation. A *reactive* solution is supposed to act, creating, updating, deleting or modifying artefacts of the target system. The number and importance of available artefact types and the level at which the tool is able to interact with each of them is an important measure of its capabilities.

Traceability is already present in many tools available, such as IDEs, CASE tools, RDBMS, requirements tools. The way each tool solves the traceability problem within the context of its artefacts is usually a black-box approach. By definition, traceability uses artefacts from several providers, each one with a possible different metamodel. Metamodeling and transformations are the key tools that traceability uses to provide effective coherence between artefacts of a system.

References

1. Dollard, K.: Code Generation In Microsoft .NET. Apress (2004)
2. Herrington, J.: Code Generation In Action, Manning Pub. Co (2003)
3. OMG: Object Management Group, MOF QVT Final Adopted Specification, www.omg.org/docs/ptc/05-11-01.pdf, (2005)
4. Costa, M., Silva, A. R. d.: Synchronization Issues in UML Models. International Conference on Enterprise Information System[illegible]chal - Portugal (2007)
5. OMG (ed.): UML Resource Page, www.uml.org/#UML2.0 (2007)
6. Kurtev, I., Berg, K. v. d., MISTRAL: A Language for Model Transformations in the MOF Meta-modeling Architecture, [illegible]ure Notes in Computer Science, Springer (2005)
7. Compuware (ed.): Optim[illegible]www.compuware.com/products/optimalj/ (2007)
8. Eclipse.org (ed.): ATL, www.eclipse.org/m2m/atl/ (2007)
9. Iivari, J.: Why Are CASE Tools Not Used, in Communications of the ACM, Oct.1996, Vol.39, Nr.10, Pgs. 94-103, Association for Com[illegible]g Machinery (1996)
10. Tata (ed.): ModelMorf – A Model Transformer, www.tcs-trddc.com/ModelMorf/ index.htm (2007)
11. Palo, M.: Requirements Traceability. Seminar Report, Department of Computer Science. University of Helsinki (2003)
12. Gorp, P. V., Altheide, F., Janssens, D.: Traceability and Fine-Grained Constraints in Interactive Inconsistency Management. 3rd ECMDA Traceability Workshop (2006)
13. IEEE (ed.). IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. Institute of Electrical and Electronics Engineers (1990)
14. Walderhaug, S., Johansen, U., Stav, E., Aagedal, J.: Towards a Generic Solution for Traceability in MDD. 3rd ECMDA Traceability Workshop (2006)
15. Aizenbud-Reshef, N., Nolan, B. T., Rubin, J., Shaham-Gafni, Y.: Model Traceability. IBM Systems Journal, Vol. 45, Nr. 3 (2006)
16. Oldevik, J., Neple, T.: Traceability in Model to Text Transformations. 3rd ECMDA Traceability Workshop (2006)
17. Welsh, T.: How Software Modelling Tools Are Being Used. Enterprise Architecture Advisory Service Executive Update Vol. 6 , N. 9, 2003-12, Cutter Consortium (2003)
18. Kolovos, D.S., Paige, R. F., Polack, F. A. C.: On-Demand Merging of Traceability Links with Models. 3rd ECMDA Traceability Workshop (2006)
19. Bézivin, J., Lemesle, R.: The sBrowser: a prototype Meta-Browser for Model Engineering. OOPSLA'98 Workshop on Model Engineering, Methods and Tools Integration with CDIF, Vancouver (1998)
20. Akehurst, D.: Proposal for a Model Driven Approach to Creating a Tool to Support the RM-ODP, The 8th International IEEE Enterprise Distributed Object Computing Conference, Monterey, USA, (2004)
21. Perini, A., Susi, A.: Automating Model Transformations in Agent-Oriented Modelling, Proceedings of 6th International Workshop AOSE 2005, Utrecht (2005)