UNIVERSIDADE DA MADEIRA

Master's thesis

Meta.Tracer - MOF with traceability

Supervisor: Dr. Leonel Nóbrega, Prof.
Prepared by: Denzil Ferreira, 2008301 MEI

July 17, 2009

Abstract

The following document proposes a traceability solution for model-driven development. There as been already previous work done in this area, but so far there has not been yet any standardized way for exchanging traceability information, thus the goal of this project developed and documented here is not to automatize the traceability process but to provide an approach to achieve traceability that follows OMG standards, making traceability information exchangeable between tools that follow the same standards. As such, we propose a traceability meta-model as an extension of MetaObject Facility (MOF)[1]. Using MetaSketch[2] modeling language workbench, we present a modeling language for traceability information. This traceability information then can be used for tool cooperation. Using Meta.Tracer (our tool developed for this thesis), we enable the users to establish traceability relationships between different traceability elements and offer a visualization for the traceability information. We then demonstrate the benefits of using a traceability tool on a software development life cycle using a case study. We finalize by commenting on the work developed.

Preface

When I started my Computer Science degree, I had the opportunity to work for a software company while attending to classes. From classes and work experience, I've learnt that engineering is much more than coding. As a field of science, engineering required specific activities in order to achieve the ultimate goal: a software product that fulfilled the requirements from the stakeholders. Although the software engineering activities (requirements, design, implementation, verification and maintenance) are quite straight forward, the order and how they are fulfilled changes according to the model used for software development process (agile, extreme programming, iterative process, waterfall and many others). Some give importance to the implementation activities and verification, while others to the requirements and design activities.[3]

As software projects got bigger in code, documentation and complexity of architecture, so did the necessity to keep track of the changes happening between different teams working on the same project. Having the knowledge of the change impacts a requirement or a component inflicts, aids a project manager to better estimate development efforts before actually making a supposably small change on a project. The need was there, and so an opportunity to explore model traceability. Meta.Tracer is the result of an intensive state of the art research, brainstorming and discussions.

With Meta.Tracer, I expect to be able to predict how much of a system needs to be changed, at a model level. As Hailpern B. et al.[4], I believe that Model-driven development (MDD) has the potential to be the next software development process, if we have the right tools to support it. This is my contribution for helping model traceability in a software project and visualize change impact on the software development lifecycle process.

I would like to thank Eng. Jorge Fernandes, from Expedita software company, for the opportunity to work and study at the same time, allowing me to experience the difference between theoretical and enterprise environment practices. I would also like to thank Dr. Leonel Nóbrega for his insight in modeling software, without him, this work would be lacking fundamental pieces for contributing to the Model-driven development process.

Introduction

Software development organizations face more and more the complexity of products, shortened development cycles and increased expectations of quality. These challenges are present in all the stages of the software life cycle and the tools for Model-driven development (MDD) may be the answer for the challenges.

Model-driven development has the objective to allow software development by using a higher abstraction level, the model level. For model-driven development, everything is a model, so it can be a model of an algorithm, a model of an object, a model of an architecture, a model of a complete system, or in our case, a traceability model of a software product.

We might argue that we already use models in software development, as we use them to explain how the project should be built, but a model, without the right tools is nothing more than a drawing, without any meaning for the software project other than static documentation. Questions like “what is this model for?”, “what can it do?”, “to whom does it relate to?” are all questions that can be answered by going around multiple documents in search for the answer, thus taking the longer route or using a traceability tool that keeps every relationship between every model ever designed in the software development, thus taking a faster route.

Traceability information is also regarded as measure the system quality and software process maturity[5], as it also increases the availability and reliability of the all process behind software management as it‘s mandated by many standards, such as MIL-STD-498, IEEE/EIA 12207 and ISO/IEC 12207. With complete traceability, more accurate costs and schedules of changes can be determined, rather than depending on the implementation team leader to know all the areas that will be affected by software or requirements changes. Keeping track of all changes and their impacts on a small project might seem feasible but if we take into account multiple teams working, product family lines and multiple sources of documentation, this task soon become unthinkable for one person alone.

Some tools have been developed over the years, in order to give some organization to this entropy. CASE tools were built to facilitate the various stages of the software life cycle but they failed because the stages are not well integrated (even within the tools of a single vendor), leading to the “silo problem”. A software documentation uses so many different kinds of artifacts, each one of them with their own models and languages. Would it not be great if there was some sort of common language upon which all models were built on? It is here that our window of opportunity is open.

If the all software development world used the same language and standards, tools from different vendors have a chance at interoperating. So came along eXtensible Markup Language (XML). It was meant to provide the flexibility in specifying documents and allow them to be exchanged. But by lacking a standardized specification, different interpretations of the XML have produced non-interoperable tools. Not only must tools interoperate, but broad support for traceability and inconsistency management between and among different models and artifacts is essential.

Another aspect of traceability is its granularity and perspective. As models can be a set of abstractions that together form the description, requirements and even implementation of a software system, we need to think about the limitations that many modeling tools present, such as information about the level of detail

depicted on the model itself, and also the perspective intended for the model, which changes frequently and how it handles multiple versions of the same models.

Traceability tools need to address these limitations and reduce them. They could combine a set of tools, designed to help the user to better specify requirements, model solutions and produce more and faster. As stated by Robert Brcina et al.[6], traceability can support the decision making by facilitating the software comprehension, the change impact analysis and the minimization of risks in software development, but it is only beneficial as long as the information on traceability is not wrong. If it is, bad decisions and the introduction of errors are a consequence. We can not guarantee that the information our traceability solution provides is correct. It all depends on the information that we, as users of the Meta.Tracer program provide. If we give the wrong information, evidently the information we shall get will be wrong altogether!

We advocate Model-driven development approach regarding the usage of models for describing a problem and also exploring and prescribing solutions, meaning that traceability plays an important role on the whole software life cycle. An effective traceability tool should contain semantics of the artifacts and models used, and the traceability links themselves need to be defined precisely enough and yet flexible enough to satisfy any model in any language.

Integration among the various environments and tools that need to share traceability information for better support software development must be increased. By following OMG stands for specifying models and meta-models, we allow traceability information to be available on all the tools that follow the same standards.

After a traceability model is generated, the main interest for the user is to have simple and quick access to the information he needs. Therefore, is it also our objective to facilitate this need by allowing him to visualize the traceability information.

General Concepts

In large software development companies, where software projects evolve at a rate that its hard for project managers to keep track of every change during the software life cycle, its crucial to have good documentation. Besides documentation, the project manager also needs to know where to find exactly what he needs, when he needs it (e.g. a modeled architecture or the implementation code). But documentation is not something that only the project manager has to consider.

When developers write its code on their chosen text editor or IDE, attempt to compile the code, make changes, test the changes until their programs work, they go back again and develop some more, until they achieve their final product. A problem arises when the developer adds decisions and assumptions directly on the code, since they remain hidden on the source code, making it harder to document, evaluate and trace for problems. As decisions and documentation get thrown away or hidden because, at the time, we think we won't need them, they tend to be forgotten over time.

These decisions are critical for the success and long-living of a software product. A customer may demand the record of these decisions or it may be necessary for a software quality certification. Also, if the company operates in several working sites, it is necessary to keep all of them updated. But of course, maintaining an updated software documentation is far from an easy task. Programmers are focused on meeting deadlines and it takes them away valuable time.

Everyone wishes the information is available when questions arise about why some concept was included or excluded or tested or not tested, but collecting and maintaining this data costs time and money. How should we be able to describe and preserve the relationships between the documentation artifacts of a software project?

Software development enterprises usually keep all the models, papers, code in files, stacked on a storage facility (or where ever they store them). Let's imagine that when we carry the documents to the storage facility, we tripped and all the paper models get mixed up with the models already there. It would require an huge effort to get all the papers sorted out again. The artifacts of documentation, although apparently linked to each other, are in reality several documents scattered and not related.

One alternative to several documents scattered is using a single source to electronically preserve this information. This approach could solve the links between documents but it doesn't eliminate interrelationships among concepts or artifacts. If this information is located on one single source, it will be easier to maintain and update accordingly.

Traditional documentation procedures store every model perspective and view as separate documents, all independent and at the same time related with each other, and this requires consistency management. One model, different views or perspectives may be the answer we have been looking for.

When using Model-driven development, is should be easier to communicate with the different stakeholders and teams during the software development life cycle. We should be able to link related artifacts and use it as communication medium between participants in the project.

The interrelationships between multiple types of models, and potentially different modeling languages, makes it harder for a stakeholder to fully understand the impact of a supposably small change on all the related artifacts, because artifacts resulting from any stage in the lifecycle of software development can

impact those produced at any other stage, so knowledge of different model technologies and terminologies must exist at each stage. Taking this into account, a traceability tool would allow any stakeholder to round-trip between the models, not only enabling him to have an overview of the system, but also understand how the models are related to each other. Also, if the software development artifacts are consistently related, it should enable better adaptation of new members of the development team, since they could go back and forward around the gathered information.

Even if the Model-driven development intends to raise the level of abstraction at which developers operate and, in doing so, reduce both the amount of developer effort and the complexity of the software artifacts that the developers use, there is always a trade-off between simplification by raising the level of abstraction and oversimplification, where there are insufficient details for any useful purpose.

Another issue we need to think about is the representation of the models themselves. Most models are designed in the industry relying to Unified Modeling Language (UML[7]) models. Unfortunately, UML elements can be used with enormous flexibility, causing that the same element can be used by one developer with a different meaning than to another. Domain-specific models are meant to solve these problems of ambiguity, by relying on elements that are specifically designed for a particular domain.

In general, the higher the level of abstraction a developer uses, the more choices exist for how to realize the abstraction in terms of executable code. As there can be multiple representations of artifacts inherent in a software development process, representing different views of or levels of abstraction on the same concepts, redundancy is also a problem.

Standardization provides developers with uniform modeling notations for a wide range of modeling activities. As models are to be used, read and understood by different stakeholders, each with their own backgrounds on the domain, they must understand how a change on their artifacts relates to or impacts other related artifacts that could be described in different notations from the ones they use every day.

The more models and levels of abstraction that are associated with any given software system, the more relationships will exist among those models. The round-trip problem occurs whenever an interrelated artifact changes in ways that affect some or all of its related artifacts. The worst forms of the round-trip problem generally occur when changes occur in artifacts at lower levels of abstraction, such as code, because inferring higher-level semantics from a lower-level abstractions is much more difficult than generating lower-level abstractions from higher-level ones.

Considering the challenges that model traceability imposes and attempting to solve them, demands that we learn what is model-driven development and their stand regarding modeling and software engineering. As the name implies, model-driven development is based on models and meta-models. The difference between meta-models and models and how they relate to each other and towards model-driven developmentneeds to be understood, since they are our tools for describing a software problem and achieving a solution.

One of the identified problems was the redundancy problem. This problem can be handled with model transformations, allowing us to “transform” one model into another, automatically. There has been some work done in this field and we discuss this method in further detail later on.

A question that might arise is: “What language do I use for describing my models?” and so we contrast between two approaches used by the software development community, UML and DSL languages. There was a good discussion about this issue at IEEE Software July/August 2009 magazine “Domain-Specific Languages & Modeling”, much due to the importance of providing an answer. More important than

choosing a language to describe a model is also knowing how do we build these languages, how do we store them and exchange them for later use.

OMG used MOF as meta-model to build and describe the language UML. This means that at the core of every UML based language, there is MOF core. So MOF is to us, the building blocks for any modeling language. Putting rules upon UML models is done using OCL (Object Constraint Language), which is also another contribution by OMG. Further contributions by OMG regarding MOF is discussed at MetaObject Facility and OCL sections.

In 2005, OMG's XML Metadata Interchange (XMI) was accept as an industry standard for specifying models and interchange format. Thus it is reasonable to consider XMI to store a traceability meta-model and models. Traceability information needs to be standard, otherwise cooperation between model-development tools is more difficult. The importance of standards is evident in the state of the art.

Traceability methods have been developed since the 1970's [5], starting with cross-referencing, much as the links we have on a webpage, relating documents to each other. Other techniques like matrices [8,9], databases [10] and graphs [11] were also developed. These methods contributed to achieve a certain level of traceability among software life cycle artifacts, but so far it has been difficult to exchange information between the tools developed. Each tool solves a small piece of the traceability problem and might be effective at what they intend to do, but that information is only useful on their own tool.

Traceability in Model-driven development can indeed be resolved using separate tools, but they need to be able to communicate between each other. So we present our contribution towards this goal, a solution for traceability information. We start by describing our traceability meta-model and how it was built.

To test our traceability solution, we did a case study, demonstrating the importance and benefits of traceability information on a software project artifacts.

Model Driven Development

Model-driven development requires us to shift from thinking about the implementation of a system to how do we describe a problem and model a solution. The process of developing a software system relies more on models and abstract thinking than code reuse. The Model-driven development relies on using a set of models, each of them requiring a particular set of skills to produce and evolve effectively. In raising the level of the developer abstraction to models, Model-driven development enables specialists to work with abstractions that better suit their tasks and expertise.

As models present different perspectives of an abstraction, so developers have different perspectives about the models themselves. The Model-driven development community can be divided into three groups [12]:

- Sketchers: focus on the use of UML to facilitate the understanding of code.
- Blueprinters: draw the analog between software architecture and building architecture. Create detailed design models, which are then handed off to (less expensive) coders to produce implementations. Design experts focus solely on complex design issues.
- Model programmers: support the use of UML as a development language with executable semantics, using action semantics and state charts.

Each group point of view stands correct because they use their own methodology according to their needs and demands. Since there are no strict rules on how to use model-driven development, we use what best fits us, on any particular situation.

We share the view of model programming community, being this view also shared by OMG vision of Model-driven architecture [26] (MDA), since MDA developers work predominantly with UML as their development language.

Developing a software solution is no longer adopting a technology or a programming language, but the combination of separate models, designed and structured so that together, allows us to achieve a better and working software solution, optimally without requiring countless hours of programming efforts.

Model-driven development tries to integrate all the activities of software engineering. Every artifact that arises from exploring a software engineering solution needs to be related and that information needs to be stored somewhere. Instead of having a multitude of documents scattered in file archives and in paper, we intend to help software engineers organize their software engineering artifacts by offering a tool that helps them establish traceability links.

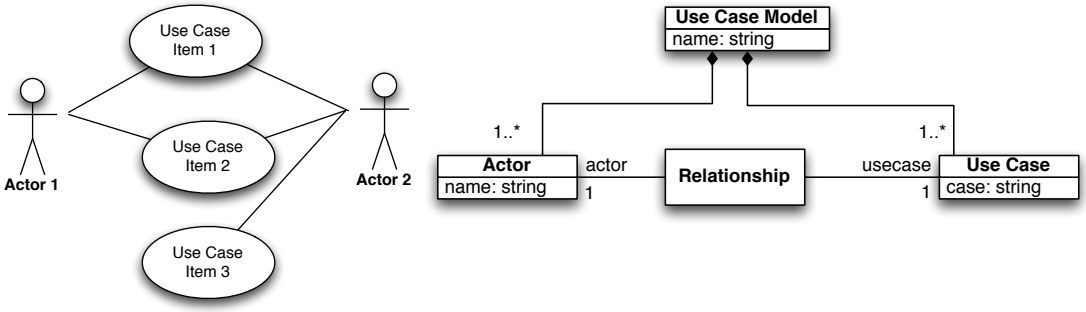
Another benefit of Model-driven development is the opportunity to automate both the creation and discovery of traceability relationships, and to maintain consistency among the variety of models used. The formality or semi-formality of models makes it possible to apply analysis methods [5,9,13,22], which may then serve as a basis to automate traceability.

This and other tools intended for Model-driven development need not only interoperate, but to broad support traceability and inconsistency management between and among different models and artifacts. By enabling developers to work at a higher level of abstraction, it will reduce the complexity and increase the understanding of a software system under development, leading to a improved final system.

Meta-models

A model in Software Engineering is an abstraction of piece of software product (e.g. requirements, code, etc). A meta-model is again another abstraction, highlighting the properties of the model itself. The relationship between a model and its meta-model is the same between a computer program and the grammar of the programming language in which it is written.

A meta-model is, according to OMG, a model that defines the language for expressing a model. Here is an example of a meta-model (**Picture 1**) for describing an use-case model:

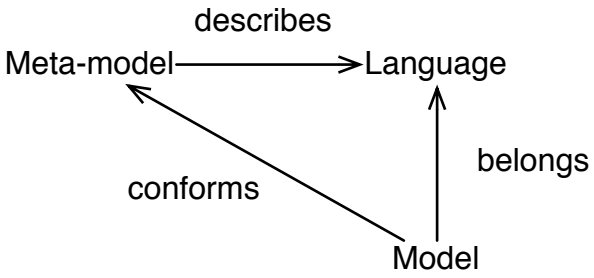


Picture 1: On the left we have the model, on the right the meta-model.

As we can see in **Picture 1**, the meta-model can be explained as: a Use Case Model, which has a name, is composed of Use Cases and Actors. Each Use Case has a name and is related with an Actor by one and only Relationship. The Actors have a name. With the meta-model on the right, we can construct Use Case Models like the one on the left. Evidently, this meta-model is far from complete and we are using it only to exemplify, in a simplified way, how we can specify a meta-model, upon which we can construct models.

The meta-model describes each element of the model and how they are related. They provide information about what attributes should be expected from the models and their types. Also, information about the cardinality of the elements and between the elements is specified.

So, schematically, we have the following relationships between the meta-models and models (**Picture 2**):

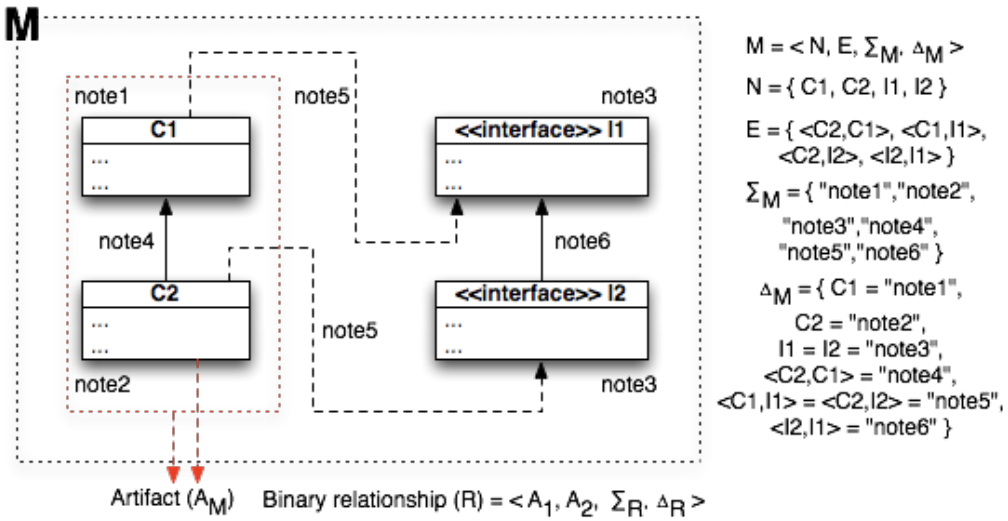


Picture 2: Mega-model as described by Jean-Marie Favre[14] and Jean Bézevin[13]

Meta-model describes the language to which the model belongs and conforms to with what's in the meta-model[13],[14].

Models

A *model* **M** is an abstraction over some (part of a) software product (ex: requirements, specification, design, code, test, ...). There is a great number of models, each one applied according to the domain the software is built. These are models that might be built upon domain-specific languages. Formally speaking, a model is a group of artifacts, interconnected with other artifacts or even models:



Picture 3: Formal model taken from Hailpern B. et al; [4]

On **Picture 3**, we have a model of a class diagram, on which we identify what is an artifact, a node (a single artifact), edges, labels and annotations, which are quite common on any model.

If we represent the model as **M** = < N, E, ∑M, ΔM > being:
M - Model itself
N - Nodes (or single Artifacts)
E - edges connecting nodes
∑M - collection of annotations present on the model
ΔM - collection of labels

An *artifact* (**A_M**) is a set of model elements of **M**. It is a complete, consistent and legal subgraph of **M**. Meaning that an artifact could represent a complete statement in a programming language grammar or a legal UML class diagram.

The set of annotations (both at the model level (ΔM) and the relationship level (ΔR)) is called meta-data. Meta-data is data about data, hence one could include the relationships in the meta-data, because relationships are links between existing model subgraphs. It all depends on what is the “base” data and what is commentary in the data. Annotations can represent both static and dynamic properties, and both functional and non-functional properties.

A *relationship* **R** maps artifacts in one model, M_1 to artifacts in another model M_2, in which M_1 can be equal to M_2. There are several kinds of relationships [4]:

- Instantiation - Nodes in A2 are specific instances of “class/type” nodes in A1.
- Refinement - Nodes in A2 represent a more detailed description of nodes in A1.
- Realization - Nodes in A2 represent an implementation of nodes in A1.
- Specialization - Nodes in A2 are specific instances of “generic” nodes in A1.
- Manual - The relationship was created by the actions of a human being.
- Generated - The relationship was created by the actions of a program.
- Derived - Nodes in A2 are a logical consequence of, and generated from, the nodes in A1.
- Implied - The relationship can be deducted by applying a set of rules.

Although these relationships might be enough to give traceability on the majority of the models, that might not be true for domain specific languages. So, in order to tackle this problem, our traceability relationships can be established and defined when creating a traceability model.

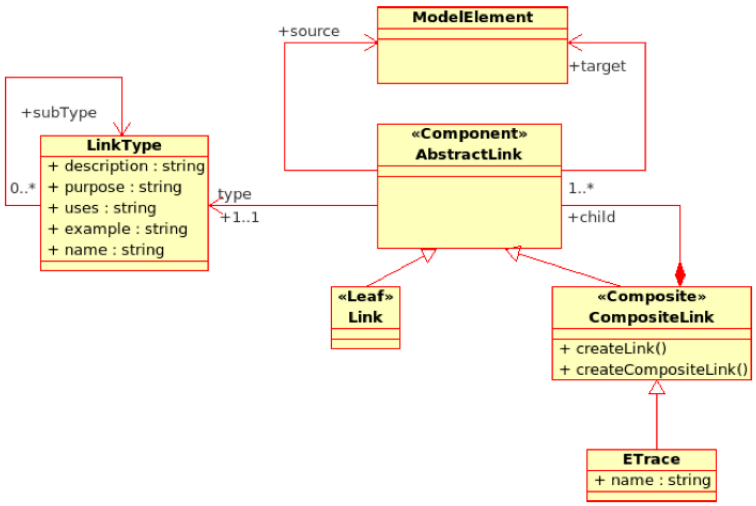
Transformations

We can define transformations as systematic (manual or automated) modification of a model and its set of affected relationships. A transformation could change a model into a new model, constrained by its current relationships, or it could leave one model unchanged and instead create new models or new relationships based on the existing ones. *Reengineering* refers to a set of changes that adds to or changes the functionality in the system. When a more systematic, structured set of semantics-preserving changes is engineered, it is termed *refactoring*. Keeping track of the changes is called *versioning*. The context of models and relationships also allows us to define reverse engineering to be the extraction of a higher-level model from another, lower-level model (or representation). Examples of reverse engineering include extracting architecture from code or extracting requirements from an architecture.

In a integrated Model-driven development set of tools, a tool supporting transformations could provide more information for the creation of traceability links generated automatically from code generation or reversed engineering.

Amar B. et al.[15], have been working on a model transformation traceability engine in a model-driven process. In software engineering, traceability has two main semantics, depending on the context[16]:
- Traceability in requirements engineering (*end-to-end* traceability)
- Traceability in model-driven development:
    - Traceability of models during transformations
    - Traceability in meta-modeling of traces and their potential uses

Their approach deals with the traceability during transformations, while we deal with traceability in meta-modeling of traces and explore its potential.



Picture 3: Amar B. et al.[8] nested trace meta-model

Their work focus on keeping information on the evolution of the models during various transformations. For each transformation, a trace model is associated with the resulting model. The importance of traceability in

model transformations is highlighted by managing transformation history information. Future work proposes using a graph-based visualization for this transformation between the transformations and the models.

We have several companies defending their own standard for transformations. Eclipse[17] effort in model transformations has been prominent with Eclipse Modeling Framework (EMF [48]) and ATLAS Transformation Language (ATL[18]). Object Management Group (OMG) [6] is contributing with two standardization efforts for realizing model transformations: the MOF 2.0 Query/Vlews/Transformations (QVT[19]) and the MOF Model-to-Text Transformation (MOFM2T[20]) languages.

There are two types of transformations which have particular interest in traceability, which are model-to-model (M2M) and model-to-code (M2C) transformations.

Kovse J.[21] claims that XMI can be used as an implementation strategy for model-to-model transformations, by expressing a specialized transformation as a series of XMI elements (<XMI.difference>, <XMI.delete>, <XMI.add> and <XMI.replace>) and apply it to an existing model to obtain the successive model.

He also argues that for M2M transformations, XMI supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and meta-model information and that it supports the exchange of differences between two models. For this purpose, it provides four elements: <XMI.difference>, <XMI.delete>, <XMI.add> and <XMI.replace>. These can be used for the following scenarios:
- Model differencing: The differences between the new and old model (both of which already exist) have to be evaluated;
- Model merging: The new model is constructed by applying a series of differences to the old model, provided that the old model and the difference information already exist.

Although this approach might seem reasonable, using XMI and XSLT for model-to-model transformations doesn't take into account the language on which the models were constructed. Because of this, QVT should be the wisest choice to make, transformation wise, since it supports model-to-model transformation and model-to-code transformation, following in return, OMG standards.

Transformations can be unidirectional or bidirectional. Updating unidirectional transformations require the means to identify the existing target-model element related to a given source-model element. XMI provides Identifiers to uniquely identify elements, inside the models themselves (XMI:ID and XMI:UUID) and between models (XMI:URI). By using these identifiers, we are able to provide to transformations mapping between source and target models.

As stated by Aizenbud-Reshef N. et al.[5], although there has been progress in the area of model transformations, the integration with traceability is still weak. We want to help in this field. A necessary step forward toward better integration among tools is the definition of a standard traceability meta-model.

Modeling languages

Right now, we know now what is a meta-model and a model and that we can apply transformations on them. But exactly how do we describe them? What language do we use? Being a model an abstraction,

this abstraction has to be described using a language. What is the language that we should use for model-driven development?

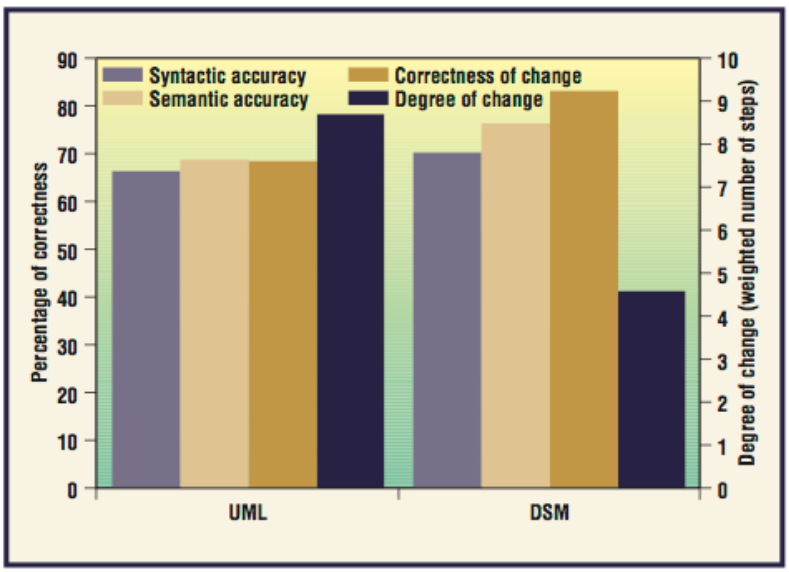
This year, at IEEE Software July/August 2009 magazine "Domain-Specific Languages & Modeling"[22], we become aware of the discussion about which is the best approach for Model-driven development, general purpose models (using UML) or domain-specific models (using DSL's).

One of the questions addressed was if are domain-specific models easier to maintain than UML models? Cao L. et al[23] argues that the wide acceptance of using DSL is due to the fact that the claims in increased productivity and ease of understanding are yet to be verified by intensive testing in software engineering.

There are two main benefits for using DSL's. The first is the improved programmer productivity. The ability to enhance systems is always constrained by how long programmers take to figure out what the code is intended to do and how it does it. The second benefit is the communication with domain experts. Software development requires communication between developers and stakeholders. A good DSL can describe a system behavior in ways a domain expert can understand.

Model syntax defines a language or a representation form and structure. Domain-specific models represent the problem space by mapping modeling concepts to domain concepts. The modeling language incorporates the business rules representing the domain knowledge. Also, these domain concepts represent the system design by specifying a system static structure and dynamic behavior. As a result, models created with a domain-specific language match well with domain specialist vocabularies, hence improving the understanding of the semantics more accurately of domain-specific models rather than of UML models.

The accuracy upon which a designer understands a model syntax and model semantics in UML and DSL models was accessed by Cao L. et al [16]. Software engineering testing was done with 64 senior IT students, by comparing their ability to understand syntactic and semantic accuracy, correctness and the degree of change required to model an existing system by introducing a new requirement (**Picture 4**).



Picture 4: Comparison of comprehension and changeability (UML vs DSM) [16]

In the end, they came to the conclusion that DSM (Domain-Specific Models) offer better comprehension as model changeability. Of course, this is a debatable conclusion. If an approach suits the problem at hand,

we use it. Standards and conventions come and go and change. If we find that UML abstractions can not provide the expressiveness we need, it's probably because a DSL is trying to get our attention[24].

Unified Modeling Language (UML)

The Unified Modeling Language is used to specify, visualize, construct and document the artifacts of a software system. From conceptual artifacts like business processes, system components and activities to concrete artifacts like database schemas, state charts and activity models, UML combines practices from data modeling concepts used with all processes, throughout the software development life cycle and across different implementation technologies. It aims to be a standard modeling language which can model every system and is widely used in the software development community.

There is currently many UML tools available in the market that support UML 2.x, but since OMG doesn't provide any test suite for compliance with its specifications, the tools makers apply their own interpretation of UML language.

A system can be modeled using several abstractions. Each abstraction will require a model diagram that best describes it. A diagram is a partial graphical representation of a system's model. They represent three different views of a system model:

- **Static** (structural) view: class diagrams and composite structure diagrams. They emphasize the static structure of the system using objects, attributes, operations and relationships.
- **Dynamic** (behavioral) view: sequence diagrams, activity diagrams and state machine diagrams. Emphasis on showing collaboration among objects and changes internally of objects.
- **Interaction** (interactive) view: communication diagram, interaction overview diagram, timing diagram. Models the flow of control and data among the system elements.

These diagrams can be constructed using any of the UML element types available by the tool used for modeling. UML does not restrict UML element types to a certain diagram type, which means that every UML element may appear on almost all types of diagrams, although this flexibility has been partially restricted in UML 2.0.

Although considered the Unified Modeling Language, UML is frequently criticized because of[25]:

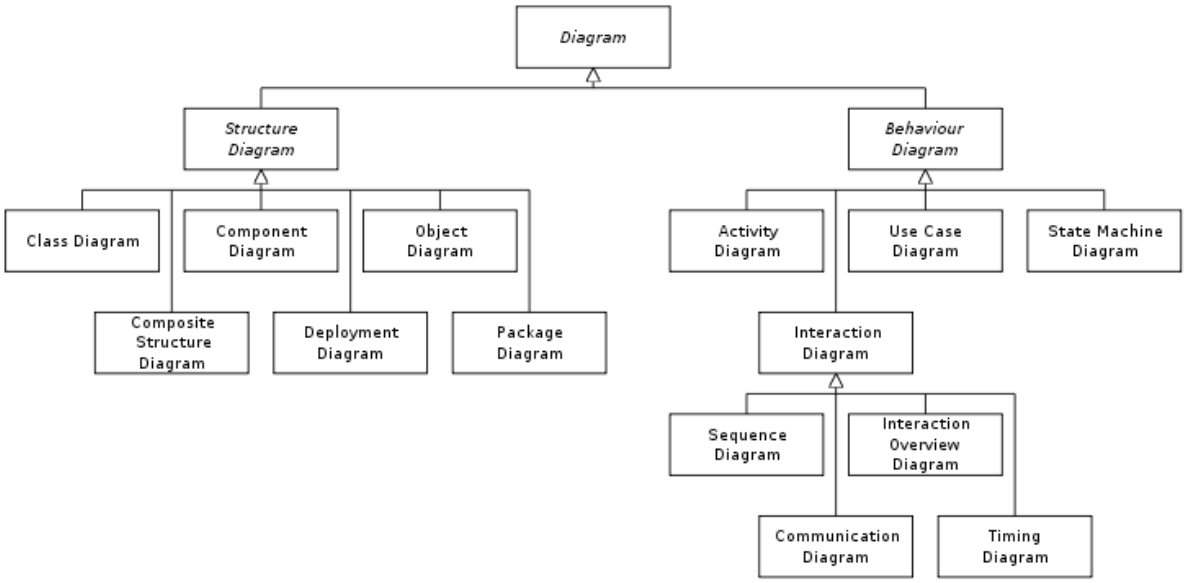
- **Extreme language size**: contains many models and constructs that are redundant or not frequently used.
- **Visually weak**: graphic diagrams that are very similar, the same relationship connector has different meanings from model to model.
- **Misused symbology**: CASE tools usually present the symbols and do not perform any kind of validation on their correct usage.
- **Lack of support for code synchronization**: if we generate code from the UML model, the code continues to evolve, leaving the model outdated.
- **Inconsistent**: Although supporters argue that the symbols look the same in order to provide uniform models, using the same symbols with different meanings is due to misinterpretation.

- **Profiles and Stereotypes as means for extensibility**: By aiming to be a general purpose modeling language, it tries to be compatible with every language required, and yet lacking consistent support for all of them.

- **Support for interchange format**: Defining a UML 2.0 model in one tool and then importing it into another tool typically leads to loss of information. UML Diagram Interchange specification lacks sufficient detail to facilitate reliable interchange of UML 2.0 notations between modeling tools. As UML is a visual modeling language, this is a drawback for modelers, that do not want to redraw their models all over again.

Neither less to its critics, UML is widely used by Software Engineering developers to specify their systems and serve as guideline for development.

UML 2.0 is composed of thirteen diagrams divided into three categories. Six diagrams represent the structure application, seven represent general types of behavior, including four that represent different aspects of interactions[26] (**Picture 5**):



Picture 5: Unified Modeling Language 2.0 hierarchy

Greenfield J.[27] argues that although UML 2.0 is a useful modeling language, it is not an appropriate language for Model-driven development, much due to its language size, making it useless for automation. Constructs are semantics-free, complicating the correct usage of UML extensions, reducing their expressiveness and limiting the ability of tool vendors to provide reliable, consistent model technologies. A model, without the right tools is nothing more than a drawing, without any meaning for the software project other than static documentation.

Rather than relying on UML alone, Greenfield J. in their Software Factories approach [20] is based instead, on the use of special-purpose, domain-specific languages (DSL's). The use of DSL's allows domain experts to be involved in the development of a software system, and according to Fritzsche M. et al.[28], this increases the quality of software as the domain requirements can be taken into account more directly and accurately.

Domain Specific Languages (DSL's)

Domain-specific languages are used in order to enhance quality, flexibility and timely delivery of software systems, by taking advantage of specific properties of a particular application domain. As we can have generic and specific approaches to solve a problem, domain-specific languages focus on achieving a solution that is optimal on a specific problem. It is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain[29].

Using a DSL approach involves both risks and opportunities. The **benefits** of using DSL's include:

- Allow solutions to be expressed in the idiom and level of abstraction of the problem domain. Domain experts can understand, validate, modify models.

- Enhances productivity, reliability, maintainability and portability, as long as we stay on the same domain.

- Embodies domain knowledge, so it enables conservation and reuse of this knowledge.

- Allows validation and optimization at the domain level.

- Improves testability if we keep on the same domain and set of problems.

The **disadvantages** of using a Domain-Specific language are:

- Costs of designing, implementing and maintaining a DSL.

- Costs of teaching a DSL to a new developer in the software development team.

- Limited availability of the DSL.

- Difficulty of balance between domain-specific and general-purpose language constructs.

DSL's can be created by extending UML language using UML Profiles. Due to it's flexibility, many DSL's have been created, leading to an higher difficulty to create tools that work with all of them. If these languages were built by extending MOF, these problems would be minimized. Using MetaSketch language workbench tool, DSL's modeling languages could be created and maintained, using XMI specification format, making it easier for tool cooperation.

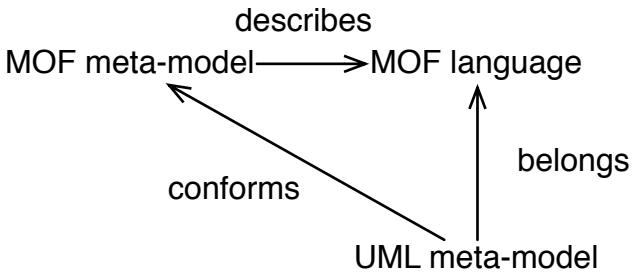
Like Sprinkle J. et al [16] states, language workbenches support DSL creation not just in terms of parsing and code generation but also providing a better editing experience for DSL users. Although they are in their early days, if their potential is realized, they could change the way we see programming today.

OMG Standards and Technologies

OMG (Object Management Group), originally aimed at setting standards for distributed object-oriented systems, and is now focused on modeling (programs, systems and business processes) and model-based standards. They only provide specifications, not implementations. Their major contributions of Model-driven development are MOF, XMI and QVT. Together, they provide the core for Model-driven Architecture.

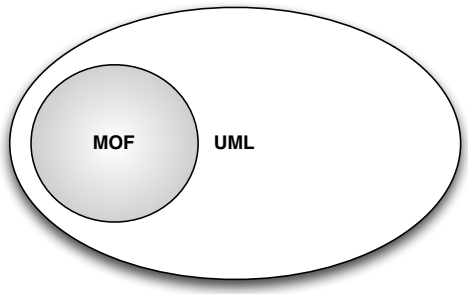
MetaObject Facility (MOF)

The MetaObject Facility (MOF) is the standard meta-model of OMG for creating models to be used in model-driven engineering. It was used to create the UML models and later reused by the software engineering community to create new languages based on UML by extending it for their personal needs (Domain-Specific models). Using the mega-model representation (**Picture 6**), we become aware of the relationships between MOF and the UML models:



Picture 6: Relationships between MOF and UML

The MOF meta-model defined by OMG describes MOF language. UML meta-model belongs to MOF language and conforms to MOF meta-model. All UML family languages include at their core, MOF language (**Picture 7**):

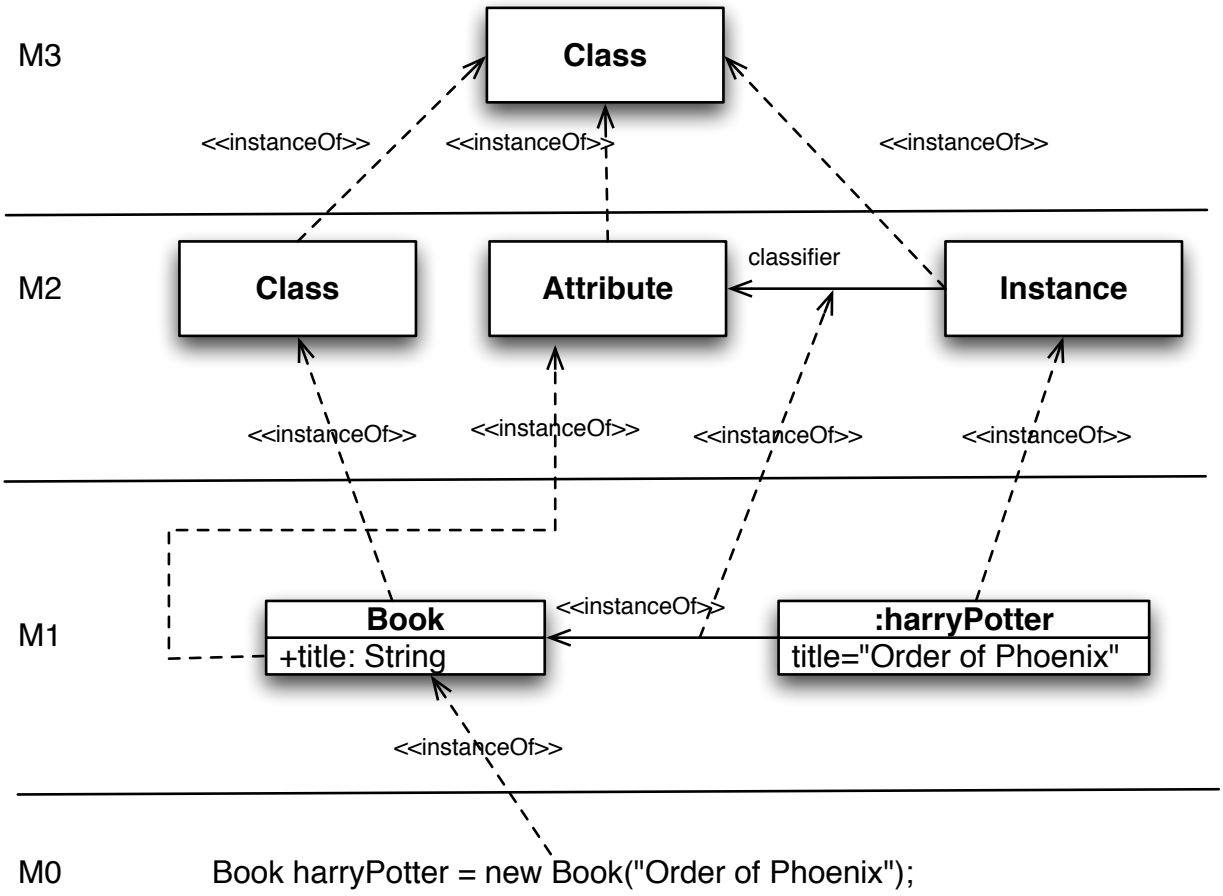


Picture 7: MOF core

OMG needed a meta-modeling architecture, and created a four-layered architecture based on MOF. It is composed of a meta-meta model at the top layer, called M3-model. This M3-model is the language used by MOF to build meta-models, called M2-models. The most common M2-model is the UML meta-model, which we discussed earlier, being the model that describes UML itself. The M2-models are used to describe M1-models. These are, for example, models written in UML. The bottom layer is the M0-layer or data layer and represents real-world objects.

Because of the similarities between MOF M3-model and UML structure models, MOF meta-models are usually modeled as UML class diagrams, since there are far more tools that support UML modeling than MOF modeling. MOF allows a strict meta-modeling architecture, every model element on every layer is strictly in correspondence with a model element in the layer above. MOF only provides a means to define the structure or abstract syntax of a language or data.

As an example of the MOF architecture, we are going to represent a Book model, using MOF and UML to describe it (**Picture 8**):



Picture 8: Example of MOF layers applied

At the M3-layer, we have Class and Element from MOF 2.0. These are instantiated at the M2-layer as elements of UML 2.0, as Class, Attribute and Instance for Class (M3). At M1-layer, we have the Book model and a model of an instance of the same model. M0 is an runtime instance of M1-model on a program.

We do not always have a M0 level, specially if we consider modeling things like Use Cases or Activity Diagrams. The M0 level are objects or realizations of the models in programs or the real world.

- The M3-layer is called the meta-meta-model, as it is the model that describes the meta-model we need for our model. It uses MOF 2.0 elements, usually from CMOF (Complete MOF). UML 2.0 uses MOF 2.0 as its meta-meta-model.

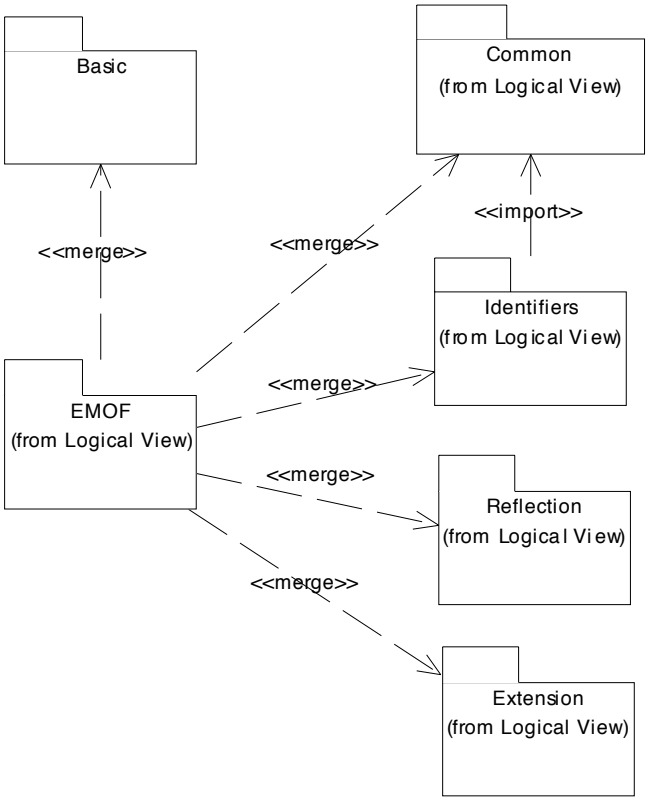
- The M2-layer is called the meta-model level. It's the model for the language described in M3-layer. Each model element at M2-layer is related with one element on the M3-layer.

- M1-layer is the model level. The elements of M1-layer are classifications of elements of the M0-layer.

- M0-layer is the instance or object level. This is the running system or object of a software system.

MOF as been defined as having two variants, by OMG:

- EMOF for Essential MOF: allow simple meta-models to be defined using simple concepts while supporting extensions. It merges Basic, Reflection, Identifiers and Extension packages to provide services for discovering, manipulating, identifying and extending meta-data (**Picture 9**).



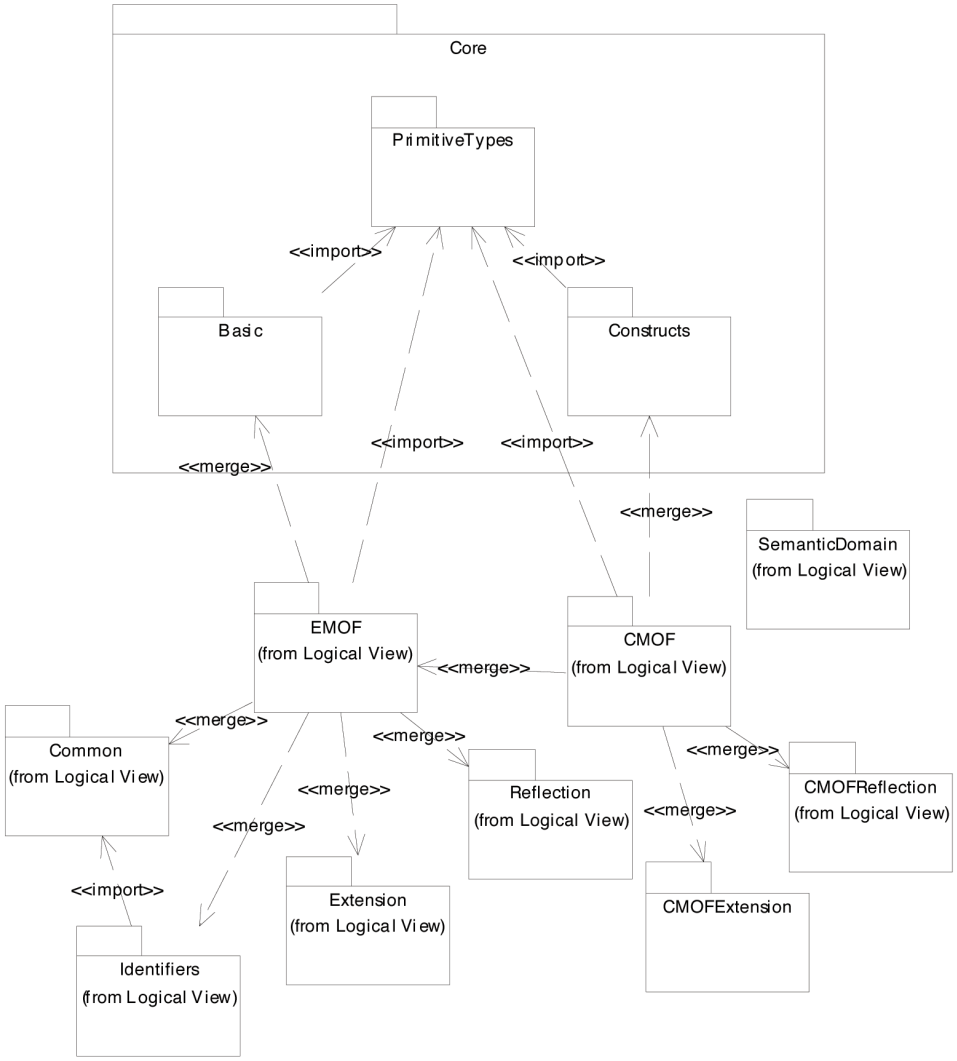
Picture 9: EMOF model overview

We could not use EMOF to define our Traceability meta-model because although EMOF defines Package and nested packages, EMOF always refers to model elements by direct object reference. EMOF never uses any of the names of the elements. There are no operations to access anything by NamedElement::name.

EMOF does not use the names of the properties, the access is done by the Property argument of the reflective interfaces. It does not matter what are the names of the Properties, the names are never used in EMOF. There is no special meaning for having similar names. The same is true for operations, there is no use of the names, therefore there's no name collision, override or redefinition of semantics.

Taking this into account, and because for traceability we need to uniquely identify traceability elements across time and space, EMOF does not provide the required elements to support it completely. Also, restricting ourselves to EMOF, meant that we did not have access to all the elements that are specified by MOF, meaning that traceability models could not be applied to all models.

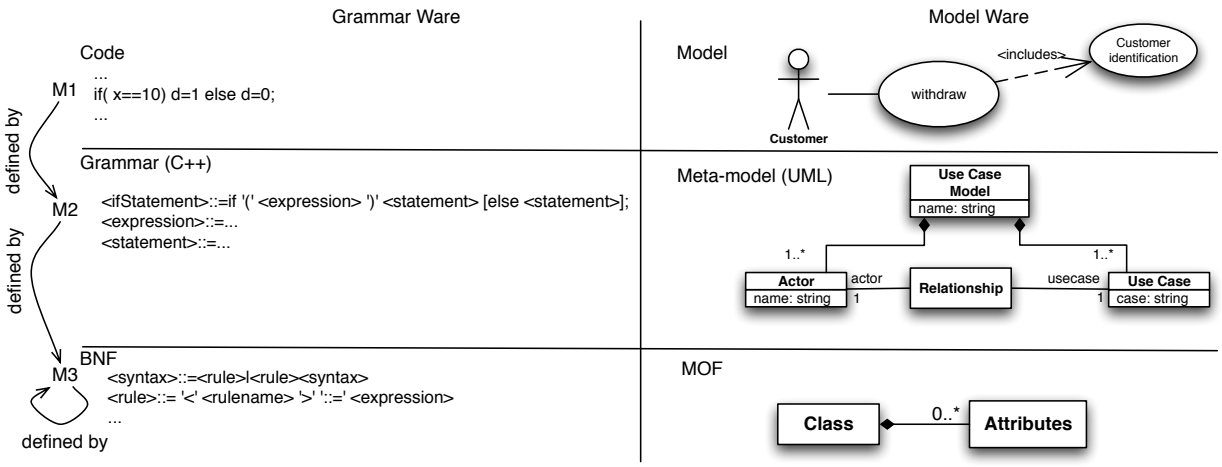
- CMOF for Complete MOF: meta-model used to specify other meta-models such as UML 2. It is built from EMOF and the Core:Constructs of UML 2. The Model package does not define any classes of its own. Rather, it merges packages with its extensions that together define basic meta-modeling capabilities (**Picture 10**).



Picture 10: CMOF model overview

CMOF allows us to model all kinds of models, since it is also the Complete MOF that defines UML. Since we want to support traceability in models based on MOF, and therefore UML models, it gives us the elements we need to express traceability links between them.

MOF is to models, what Backus Naur Form notation[30] (BNF) is for defining programming languages. Like BNF is defined with BNF, MOF is defined with MOF (**Picture 11**):



Picture 11: Relationship between BNF and MOF

On the left, we have the “grammar ware” view: BNF is defined by itself, creating the M3-layer. M2-layer is constructed using BNF, creating the grammar (in this case C++), which will allow us to program. M1-layer is the code itself. M0-layer (not illustrated) is the running code on our computers.

On the right, we have the “model ware” view: MOF is defined by itself, creating the M3-layer. M2-layer is constructed using instances of MOF, creating the meta-model (in UML), which allows us to model use-cases. M1-layer is the model itself.

For our traceability meta-model, we extend the ElementImport for TraceabilityElement, Package for TraceabilityView, Relationship for TraceabilityRelationship and Element for TraceabilityProperty elements.

Our effort was put into finding the correct MOF elements to extend our traceability meta-model. We wanted a traceability meta-model that allowed flexibility, expressiveness and semantics, and based on CMOF, so that we could apply it on all the models modeled using MOF and XMI standards.

Object Constraint Language (OCL)

The Object Constraint Language (OCL) is a declarative language for describing rules that apply to Unified Modeling Language (UML) models. OCL may be used with any MetaObject Facility (MOF) meta-model, also still including UML. It is a precise text language that provides constraint and object query expressions on any MOF model or meta-model, that cannot otherwise be expressed via graphical notation. OCL is a key component of the OMG recommendation for transforming models, the QVT.

OCL language statements are constructed in four parts[31]:

- a context that defines the limited situation upon which the statement is valid.

- a property that represents some characteristics of the context (e.g. is the context a class, a property an attribute?).

- an operation (e.g. arithmetic, set-oriented) that manipulates or qualifies a property.

- keywords (e.g. if, then, else, and, or, not, implies) that are used to specify conditional expressions.

OCL supplements UML by providing expressions that tries to disambiguate natural language or reduce the difficulties of using mathematical expressions. It allows a MOF model to be more precise by associating assertions with its meta-elements. QVT uses OCL, being a language on top of OCL.

Other use for OCL is replacing, in a lightweight way, formal specification languages like Z. In this case, it is applied on the level of application modeling, where it is used to express properties like invariants of the static class model, pre-conditions and post-conditions for operations and guards for state transitions.

The OCL meta-model is composed of two parts [31]:

- Types package: defines the types which are recognized by OCL and relates them to core UML concepts. Every type is a super type of OclVoid, which implies that the undefined value OclUndefined is an element of every types in OCL. This includes the Boolean type, which means that OCL has a three-value logic (undefined, true and false) instead of the common two-value logic (true and false).
- Expressions package: OCL defines all operations and expressions to be strict, except where something els is explicitly specified. And expression is said to be strict if it evaluates to OclUndefined whenever one of its parameters is undefined.

OCL has moved from a constraint language to a full query language for object-oriented models on its latest revision 2.0. This is mostly due to the introduction of the tuple-type. It allows to combine multiple values of different types into one value, thus being able to be transported together through iterate-expressions which allows essentially the full range of query expressions to be formulated. In addiction to the standard usage types invariant, pre-conditions, post-conditions, and guards, the specification also defines the use as initial or derived value expression, operation body expression, or definition expression, all of which use OCL as a generic query language.

Using OCL, constraints can be established for the traceability links, either internally to the models or externally. This allows us to establish constraints that are applied to any model built with a language that extends our traceability meta-model. As it also possesses query capabilities, it allows us to navigate through the traceability links established among the traceability model and related models.

XML Metadata Interchange (XMI)

The XML Metadata Interchange (XMI) is an OMG standard for exchanging metadata information via Extensible Markup Language (XML). It can be used for any metadata whose meta-model can be expressed in MetaObject Facility (MOF). It is commonly used as an interchange format for UML models, although it can be also used for serialization of models of other languages too. It is considered an industry standard since 2005 with ISO/IEC 19503:2005[32].

One purpose of XML Metadata Interchange (XMI) is to enable easy interchange of metadata between UML modeling tools and MOF-based metadata repositories in distributed heterogeneous environments, thus meaning that traceability could be also exchanged between different locations. XMI is also the medium by which models are passed from modeling tools to software generation tools as part of Model-driven development.

XMI combines four industry standards:

- XML: eXtensible Markup Language, a W3C standard.

- UML: Unified Modeling Language, an OMG modeling standard.

- MOF: MetaObject Facility, an OMG language for specifying meta-models.

- MOF mapping to XMI

The integration of these four standards into XMI allows tool developers of distributed systems to share object models and other metadata required, such as traceability information.

As XMI is the standard that OMG uses to provide the MOF specifications, it seems logic to use the same standard to store the traceability meta-model and models. This allows us to reuse the models with any tool that is XMI compliant. Using the XMI:UUID and XMI:ID identifiers for intra-model relationships and XMI:URI for inter-model relationships, we can relate model artifacts to the traceability model without ambiguity, thus we are giving the opportunity for the development of tools to further our proposal for traceability.

Traceability

Working with traceability links requires understanding first exactly what is traceability, in the context of software engineering, what is it's role in Model-driven development. As discussed earlier, UML is the most widely used language for modeling in software engineering, so we need to understand what is UML and why it isn't the best language for creating and maintaining traceability links. Since UML is described using MOF, we argue about the role of MOF in all the UML language family and why we think it's better to have traceability links at MOF level.

Since we also want to support the future development of other tools that might want to reuse our traceability meta-model, we support the OMG standard for specifications, using XMI for that purpose. Also, if we enable OCL to define constraints between artifact relationships, automatic transformations could be supported in the future. Although we mention their contribution for further traceability work, it will not be our focus for this document.

Model-driven development has as main principle that everything is a model, so it seems a good solution to store traceability information as models too. By using a separate traceability model has the advantage of keeping traceability information independent of traced models.

A trace represents a chain of relationships across different models (or artifact representations) through a software product's life cycle (ex: mapping a requirement to its corresponding architectural element, to the code that implements it, and to the test case that validates it). The property of traceability is core to the value proposition of Model-driven development.

Traceability also should give us the ability to chronologically relate uniquely identifiable entities[33]. These entities can be, for example, a natural language description of a requirement (non-model artifact) and a model of the architecture employed. This kind of traceability between non-model and model artifacts is called *end-to-end* traceability [10] and with it, we are able to show that a requirement is implemented or is satisfied in another artifact.

There are two different kinds of traceability links, according to Model-driven development, the *explicit traceability links*, which are links established in the models themselves using a suitable syntax, and the *implicit traceability links* which are traceability links generated somehow by the modeling tool. Since we are working with manually established traceability links with Meta.Tracer, we are creating explicit traceability

links, which can in the future be implicit traceability links, with further development of Meta.Tracer, by providing automatic generation of traceability links.

Aizenbud-Reshef N. et al. [5] shares with us that traceability relationships help stakeholders understand the many associations and dependencies that exist among software artifacts created during a software development project. The extent of traceability practice is viewed as a measure of system quality and process maturity and is mandated by many software engineering standards. One issue that impedes wide adoption of traceability is the overhead incurred in manually creating and maintaining relationships. How we manage these relationships depends on its nature and origin, so both manual effort and computation time needs to be invested to manage them. The goal is to minimize manual effort at the expense of increased computation resources.

Each model has its own notation, representation, tools and users. Thus developers, tools, artifacts and processes are largely isolated and only weakly integrated. The formality or semi-formality of models makes it possible to apply analysis methods, which may then serve as a basis to automate traceability.

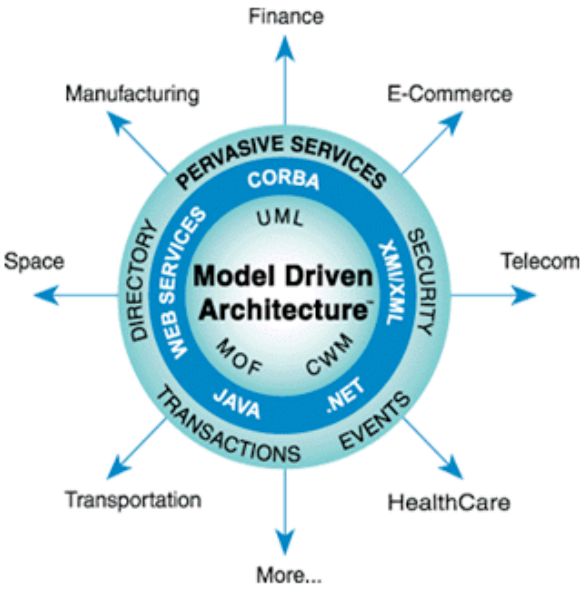
The ability to round-trip across models reflects the bidirectional nature of a trace path (you should be able to go forward and backward along a trace, without losing your way). Traceability in software modeling can be defined as the ability to follow the different model elements from the design step down to the implementation, since it represents relations between artifacts in different phases of the development process.

Traceability may be used for different purposes such as understanding, capturing, tracking and verification on software artifacts during the development lifecycle.

Traceability standardization

Standardization provides developers with uniform modeling notations for a wide range of modeling activities [4]. It also allows the development of tools for creating and manipulating models, generating artifacts (such as code) from models and reverse engineering models from other artifacts for instance.

Realizing the lack of standards on Model-driven development, OMG came forward with its own particular realization of MDD, using the term Model-driven Architecture[34] (MDA).



Picture 12: OMG Model Driven Architecture overview

In **Picture 12**, we can see that to OMG, MDA is supported by MOF and UML, and since UML is derived from MOF, we can express UML models as MOF models if we keep only what matters.

MDA is based on two models, PIM (Platform-independent Model) and PSM (Platform-Specific Model). PIM is the specification of the business functionality and behavior. This is the base model for the architecture over which different PSM models describe how the base model is implemented on a different middleware platform.

But MDA is still not widely used, due to the effort required to develop and maintain software artifacts and the lack of tools that support it. Hopefully this should change in the future.

The most interesting part of OMG contribution is not MDA by itself but the standardization of languages that came with it, namely MOF [1], XMI[9] and OCL[35].

As MOF is the language used to define UML family languages and it is the core language on all the languages based upon UML. It is our belief that traceability meta-models should be defined using MOF and extending UML, as it provides us with the flexibility, power and expressivity of MOF core elements and extends the UML family with traceability capabilities.

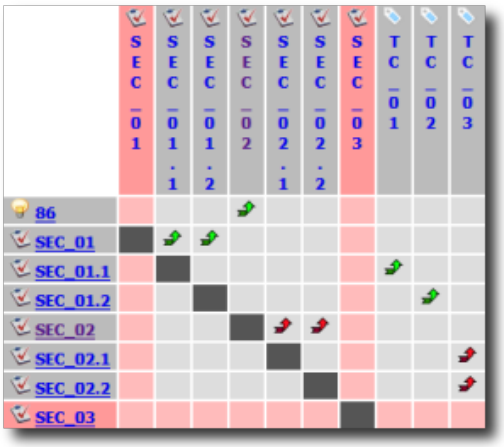
By extending MOF with traceability, we are at the same time extending all the modeling languages that are built upon MOF, covering not only UML as also DSL models, meaning that traceability relationships can be established between artifacts on any models based upon MOF.

One of the main concerns about traceability is the lack of integration among the various environments and tools that need to share traceability information. Our model provides a definition of traceable artifacts (TraceabilityElement), and is able to specify different types of traceability relationships (TraceabilityRelationship) with standardized semantics. The meta-model for traceability allows customization and supports extensibility mechanisms. In addition, we follow XMI standards, which allows us to exchange traceability information.

Only if there is an effort from all modeling tools makers to support the same standards, shall we be able to enhance the traceability mechanisms, thus contributing for a better model-driven development methodology.

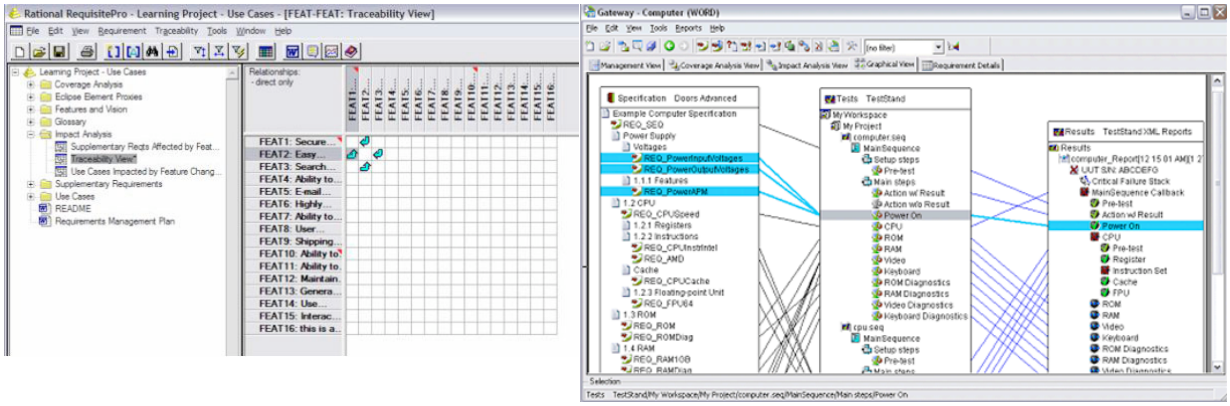
State of the art

Cross-referencing was the first traceability method in the early 1970s and it implied the usage of links between different artifacts by matrices, databases, hypertext links, graph-based approaches, formal methods and dynamic schemes. The traceability information had to be created and maintained manually, as the responsibility for managing its validity with respect to change. The traceability information quickly became outdated as the system evolved. Still, it is used today by software engineers for quickly creating dependency checks when developing[36].



Picture 13: Psoda's cross-reference traceability matrix [20]

IBM RequisitePro[37] and Telelogic DOORS[38] introduced monitoring of linked elements and did in fact indicate links that were possibly in need of modification, but provided simple links, without any semantics. Although it "supported" integration with other software development tools to other products of the software life cycle, using a specific export format. It does not provide rich traceability schemes, thus allowing only simple forms of reasoning about traces.



Picture 14: IBM RequistePro [21] on the left, Telelogic Doors [22] on the right - traceability views

Further work was done to achieve relationship semantics. There are two approaches to supporting richer semantics:

- The first approach is to allow users to add attributes to relationships. Telelogic Doors used this method, but the attributes were treated uniformly and cannot provide specialized behavior for different types of relationships.

- The second approach is to provide a predefined standard set of relationship types that can be supported by the tool. This approach was tested by Ramesh B. et al.[39] upon extensive observation of traceability practice in several organizations. Using a predefined standard set of relationship types, they allowed the users to establish traceability links between the artifacts. The problem was that they could not extend the relationship types to what was needed (e.g., Domain Specific Languages - DSL's) and keep the link information within the artifacts themselves, which "polluted" the models with information that were not part of the original model and only the tools that created the links could recognize and manipulate these links.

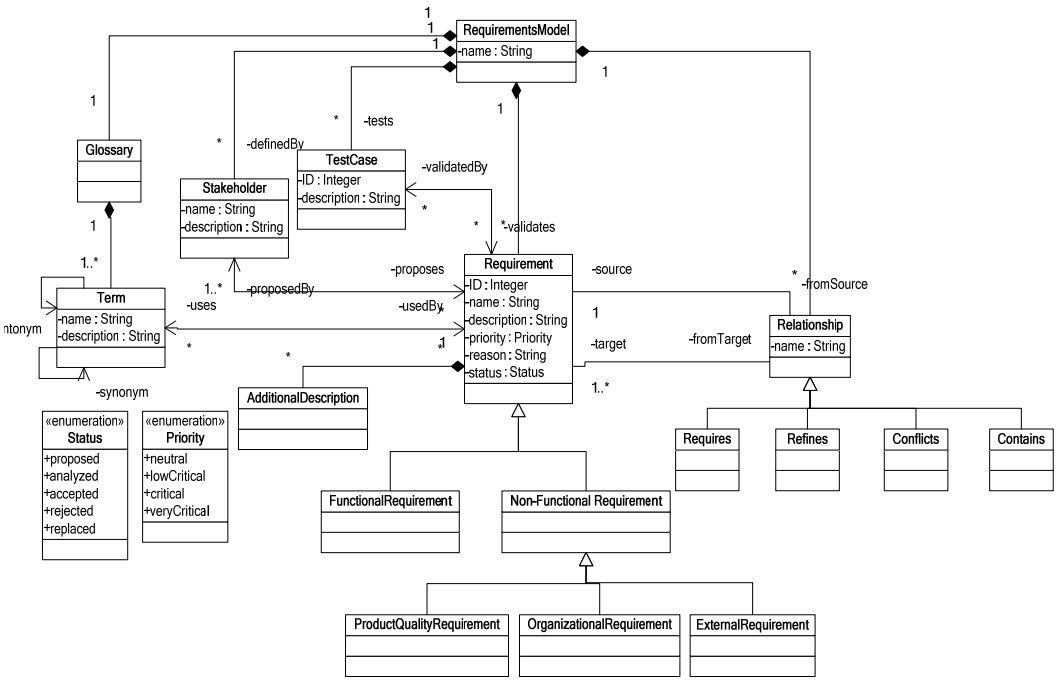
The concept of *metadata repository*[40,41], which keeps all the artifacts in one unique location is regarded by other tool developers as less appealing, since it requires tight integration and strong coupling with the system. One of the advantages of this approach is the ability to uniquely identify artifacts across time and space, as long as they stay on the same repository.

Not having to manually specify and maintain traceability has been the focus of Hayes J. et al.[42], by applying information retrieval techniques by improving recall and precision to reduce the number of missed traceability links and irrelevant links (traceability mining). By comparing their results with a professional requirement analyst, initial results indicated that the tool achieved higher percentage in detecting these links than the analyst.

Recovering traceability links where they did not exist before has been the work of Marcus A. et al.[43], by using latent semantic indexing to restore traceability links from documentation and source-code, identifying them based on similarity measures. They defined a validity scale, by using the notion of causal conformance relationships that represented logical semantic dependencies among documents with implied logical ordering over time. The conformance analysis compared node-modification time stamps and link validation time stamps to produce a heuristic conformance rating.

While these methods were applied after the models were already existent, traceability should begin much sooner. Keeping track of an ever evolving software product life cycle is for a project manager a mission that begins on requirements [14,44]. Kurtev I. et al. [30] propose the formalization of requirements traceability links

and use a predefined set of rules which are used for validation and change impact analysis. These requirements traceability links help the project manager keep requirements up-to-date, although they do not help him in determining possible impacts of requirements changes in design models but further work is being developed in that area.



Picture 15: Requirements meta-model as proposed by Kurtev I. et al.[30]

The formalized requirements relationships were defined as:

- **Requires**: A requirement R1 *requires* a requirement R2 if R1 is fulfilled only when R2 is fulfilled. R2 can be treated as a pre-condition for R1.

- **Refines**: A requirement R1 *refines* a requirement R2 if R1 is derived from R2 by adding more detail to it.

- **Contains**: A requirement R1 *contains* requirements R2...Rn if R1 is the conjunction of the contained requirements R2...Rn. This relation enables a complex requirement to be decomposed into child requirements.

- **Conflicts**: A requirement R1 *conflicts* with a requirement R2 if the fulfillment of R1 excludes the fulfillment of R2 and vice-versa.

By considering that a requirement is a property which must be exhibited by a system, the formalized requirement relationships were used to define also change impact rules. Traceability relationships need also to be formalized if we intend to have a rigorous change impact analysis and so far, there are non-existent and we look forward for some work on the area.

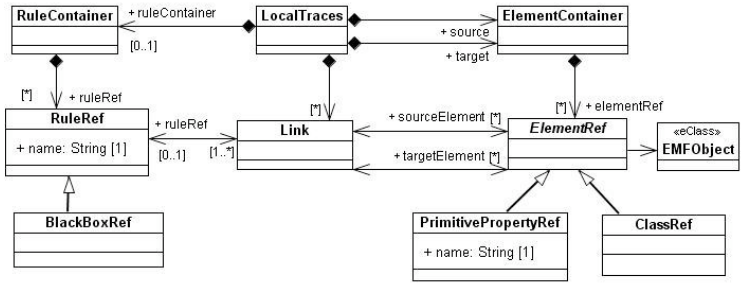
Obtaining implied traceability links from already defined links is the focus of Sherba S.A. et al.[45] by using generic services that discover, create, maintain and views relationships, allowing the user to define new derived relationship types as a chain of existing ones.

According to Aizenbud-Reshef N. et al.[5], analyzing change history may also be a source for automatically computed links. Ying A. et al.[46] and Zimmermann T. et al.[47] work on applying different data mining techniques to the change history of a code case to determine change patterns, using set of files that were changed together frequently in the past. Files in the same change set are related and can be used to recommend potentially relevant source-code to a developer changing code. Aizenbud-Reshef N. et al.[5] also makes us aware of other attempts to improve the maintenance of manual relationships, by means of event-based traceability, using publish-subscribe relationships. This would make automatic traceability changes when possible using rules and inform developers for appropriate action when not.

Drivalos N. et al.[48] defines semantically rich traceability links between models expressed in diverse modeling languages using case-specific traceability meta-models. Using EVL (Epsilon Validation Language), restrictions could be applied within these traceability meta-models and apply them as cross-model constraints.

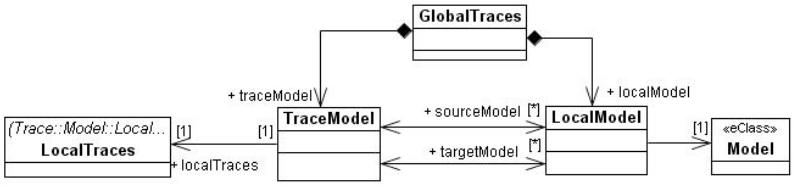
Glitia F. et al.[49] presents a framework for defining traceability using UML 2.x and Gaspard environment for creating high level models to generated code using transformations. As previously mentioned, UML 2.x lacks the semantics needed for establishing well defined traceability links, meaning that these traceability links need to be created and maintained using another meta-model, in this particular case using EMF (Eclipse Modeling Framework)[50].

Their approach for implementing traceability in a transformation chain relies on the idea of separating traceability into levels, which are referred as local (**Picture 16**) and global traceability levels (**Picture 17**):



Picture 16: Local trace meta-model

The local trace meta-model captures the traces between the input and output of one transformation.

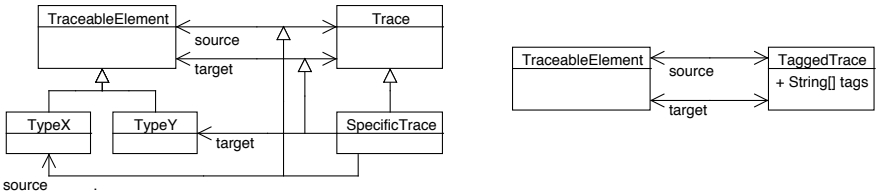


Picture 17: Global trace meta-model

The global trace meta-model links local traces according to the transformation chain. A global trace model is a main entry point from which all the local trace models are found, and describes that a target/source model of one transformation is a source/target model for the next/previous transformation.

Vanhooff B. et al[51] proposes a traceability meta-model that serves as input for model transformations. Most of the current model transformations do not take into account previous transformations. By

considering intermediate models as part of a global traceability graph, they proved that it can be used to derive useful relationships between inter and within input models of subsequent transformations. They rely on tags to disambiguate navigation on traces.



(a) pure metamodel (b) trace tagging

Picture 18: Vanhoof B. et al.[48] traceability meta-model

Anquetil N. et al[52] exposes the need for traceability in Software Product Lines (SPL's). Since software development aims to minimize development costs, SPL's engineering and generative programming have been an aspect of interest. This in return, raises the need for more intricate traceability solutions. The difficulties linked to traceability in SPL's go from a large number and heterogeneity of documents to understanding the variability consequences of leaving out a feature in order to create a derived product.

Traceability in MDD

Up until the present date, there is no correct system or method to provide traceability links between artifacts. A good traceability system/method should allow most, if not all, of the following:

- Traceability links should be generic but yet specialized enough to provide specialized behavior for different types of relationships.
- Traceability links should allow extensions for relation types according to other languages
- Analyze change history for a source of automatic computed links.
- Traceability links should be stored outside the models, in order to not “pollute” them.
- Be capable of dealing with different type of models, such as business, data, design and test, and their artifacts. The need to support collaborative and multisite development adds new types of artifacts, such as team-room documents, chat discussions and e-mails for instance.
- Keeping link information separate from the artifacts requires the ability to uniquely identify artifacts across time and space.
- Support detecting, representing, storing and propagating a wide range of inconsistencies and provide management for the inconsistencies.
- Support cross-model constraints.

The increased burden of manually specifying and maintaining traceability information is a major impediment to the general acceptance of traceability practice. Although some methods like text mining and information retrieval techniques are used to infer relationships between artifacts, the links still need to be validated by an analyst.

One of the most challenging aspects of traceability is how to maintain the correctness and relevance of relationships while the artifacts continue to change and evolve. Most of the previous work tried to apply methods that would auto-correct mistakes or errors produced on the models, many times these corrections would be wrong. This is mostly due to the nature of the models and the flexibility that abstraction brings along. The best method will be to notify the user that errors are present and propose corrections and let the user decide. After all, tools are meant to help you and not to replace you.

Traceability needs to be tackled a step of a time, and not considering the system as a whole. A well-defined, automated method of accomplishing traceability would be of value in any domain, on any project, with any methodology [4].

Our approach: Meta.Tracer

When Hailpern A. et al [4] states that they do not believe that graphical programming will succeed, because it requires more time than writing code, we argue that given the right set of tools, preferably integrated set of tools designed to support Model-driven development from the beginning to the end of the software development process, would in the end, allow better software engineering.

MetaSketch already allows the generation of code from high level abstraction models. In order to deal with the complexity of such a system, it is necessary to introduce a traceability mechanism helping both the users and the developers keeping track of elements. By extending the MetaSketch framework and using the same standards as OMG proposes, Meta.Tracer intends to provide a traceability mechanism for supporting Model-driven development.

Meta.Tracer is designed to help the stakeholder better understand the system and make informed decisions for change impacts. Being part of a set of tools designed to support Model-driven development, and using the traceability meta-model we propose, it will allow us to relate different artifacts and models, allowing him to round-trip between them.

Models that are modeled using UML 2.0 language, are in their roots, modeled using MOF (MetaObject Facility). So considering the MOF roots and following OMG[53] (Object Management Group) recommendation of storing models in XMI[54] (XML Metadata Interchange), the models can be easily reused in all software MOF and XMI compliant. This means that the traceability models that we construct with our tool would still be valid on other tools. Since we are focusing our work on providing a meta-model for traceability built upon MOF, this work can be easily extended, providing other contributions for the model-driven development process.

Until now, there isn't a standard set of traceability link types. So our approach is to allow the creation in run-time of traceability links (TraceabilityRelationship).

Traceability meta-model

According to Bondé L. et al[55], a good traceability model should be complete enough to record the necessary trace information. But a too complex model is difficult to exploit. To solve this problem between completeness and simplicity, we propose a traceability meta-model with generic artifacts, to which we add as many attributes we need to make them relevant to the domain upon which we model. Like we mentioned earlier, the ability to define specialized languages and different abstraction levels is essential to MDD.

A traceability meta-model captures how model elements may be related by trace relations and defines the semantics of such relations. It is often necessary to distinguish between different kinds of traces. For example, we might want to distinguish a trace between a textual requirement and an architectural element from a refinement within the design. The required kinds of traces are often highly project dependent. Until now, there is not a standardized set of traceability relationship types, thus we try to be flexible enough for

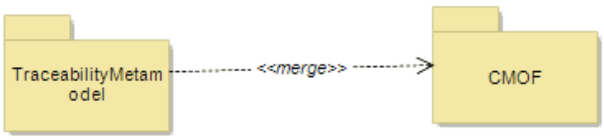
the user to define the links he needs for tracing his models, enabling the user to give semantics to the relationship using tags as needed.

At a meta-level, the set of models and relationships (including their annotations) can be constrained to satisfy a set of consistency specifications. Due to the human nature of the software-development process, inconsistency may persist and must be managed for extended periods of the software life cycle. This process of controlled chaos has been called *inconsistency management* [4].

We target our approach at MOF level. Keeping in mind that we do not want the traceability information in the models themselves, this information is built when the user is modeling and should remain hidden until the user opens the traceability model itself, being the traceability links merged with the models on-demand.

This approach was presented by Kolavos D. et al[56], and due to the fact that we build our traceability models using MOF, it allows us to map unambiguously artifacts present on the models with the traceability meta-model using the XMI identifiers.

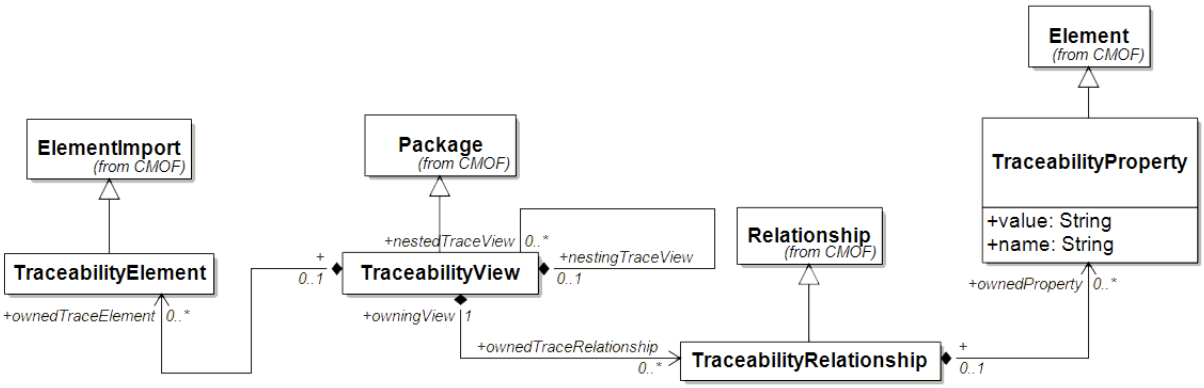
We began by extending Complete MOF (CMOF) with our traceability meta-model in order to define a language that incorporates required concepts for traceability (**Picture 19**).



Picture 19: Merging CMOF with our Traceability meta-model using MetaSketch

Drivalos N. et al. previous work [17] on traceability meta-models, a language has to be defined between each pair of models, with restrictions made to the traceability links using EVL. This is feasible when we are relating a small group of models. But what happens when we begin to relate different projects together, when they reuse models already built on another project? Inter-project traceability information is essential for enabling reuse of models, much in the way we reuse code.

As we wanted to build our meta-model as an extension of MOF, we defined TraceabilityElement as a specialization of ElementImport, TraceabilityView as a specialization of Package, a TraceabilityRelationship as a specialization of Relationship and a TraceabilityProperty as a specialization of Element (**Picture 20**):



Picture 20: Traceability meta-model (MetaTracer) defined from CMOF

A TraceabilityView is composed of TraceabilityElements and TraceabilityRelationships. Each TraceabilityView can have more nested TraceabilityViews, each with its own set of TraceabilityRelationships. Each TraceabilityRelationship can have its set of TraceabilityProperties. All of these elements are constructed from CMOF, thus this is a language built on MOF, which is the common language in all the UML family.

On the context of UML languages we are extending UML family languages with a new element that is capable of allowing the definition of traceability models between any model in any language from this family, including the language of traceability itself.

Traceability view

TraceabilityView is a specialization of Package from CMOF. We want a TraceabilityView to be able to have several other TraceabilityView inside it, which allows us to establish traceability links between several TraceabilityView as a whole. This means that we can establish traceability links inside the model itself and between other models, since for MOF, a model is a package.

Package is used to group elements, and provides a namespace for grouped elements. It is a namespace for its members, and may contain other packages. Only packageable elements can be owned members of a package. By being a namespace, a package can import either individual members of other packages, or all the members of other packages. In addition, a package can be merged with other packages.

Traceability element

TraceabilityElement is a specialization of ElementImport from CMOF. It allows us to expand our traceability language by “importing” other elements to the language, being the imports references to other model elements.

ElementImport identifies an element in another package, and allows the element to be referenced using its name without a qualifier. It is defined as a directed relationship between an importing namespace and packageable element. The name of the packageable element or its alias is to be added to the namespace

of the importing namespace. It is also possible to control whether the imported element can be further imported.

Traceability relationship

Aizenbud-Reshef N. [5] states that an optimal solution for creating traceability relationships would be to provide a predefined link meta-model, allow customization, and allow extensibility to define new link types. We also share the same point of view, and so we extended CMOF to include TraceabilityRelationship.

TraceabilityRelationship is a specialization of Relationship from CMOF. Since all MOF relationships are defined using Relationship, this means that our language can reuse all the defined MOF relationships. A TraceabilityRelationship is composed by as many TraceabilityProperty as we need. Since all the elements in the model have an unique identifier, the related elements in the relationship can be unambiguously identified.

Relationship is an abstract concept that specifies some kind of relationship between elements. It can reference one or more elements. It has no specific semantics, being an abstract meta-class. The various sub-classes of Relationship will add semantics appropriate to the concept they represent. This gives the flexibility we wanted to achieve. The specific sub-classes will define their own notation. In most cases the notation is a variation on a line drawn between the related elements. One of the sub-classes we are interested in is the DirectedRelationship, since it allows us to specify a TraceabilityElement as the source and target of the relationship.

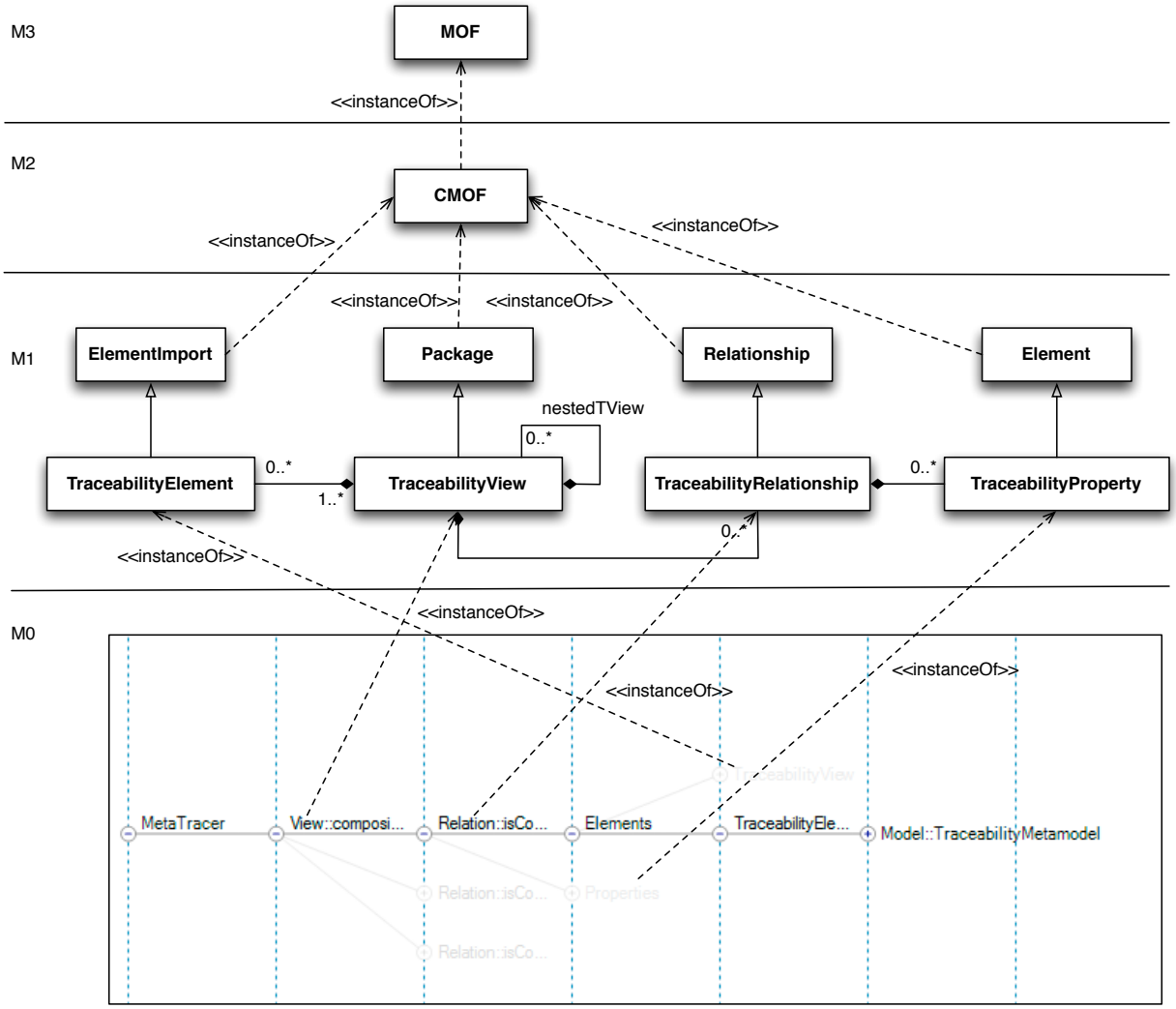
Traceability property

TraceabilityProperty is a specialization of Element from CMOF. TraceabilityProperty is regarded as an element that is owned by the TraceabilityRelationship. This allows us to add as many elements we need to the relationship itself, rather than limiting to a specific set of properties. This allows us to create generic traceability links and yet being flexible enough to add new properties to the TraceabilityRelationship for DSL's.

Element is a constituent of a model. As such, it has the capability of owning other elements. It is an abstract meta-class with no super-class. It is used as the common super-class for all meta-classes in the infrastructure library. Element has a derived composition association to itself to support the general capability for elements to own other elements. Sub-classes of Element provide semantics appropriate to the concept they represent. The comments for an Element add no semantics but may represent information useful to the reader of the model.

Traceability MOF architecture

To better understand how the MOF layered architecture is used by our traceability meta-model, see **Picture 21**:



Picture 21: Traceability meta-model MOF architecture

Following MOF 4 layers architecture, as depicted previously, we have the following:

- M3-layer: since our traceability meta-model is built extending MOF itself, we have here the meta-meta model MOF 2.0. For simplicity, we describe MOF as a class on the model, but it includes all the MOF infrastructure.

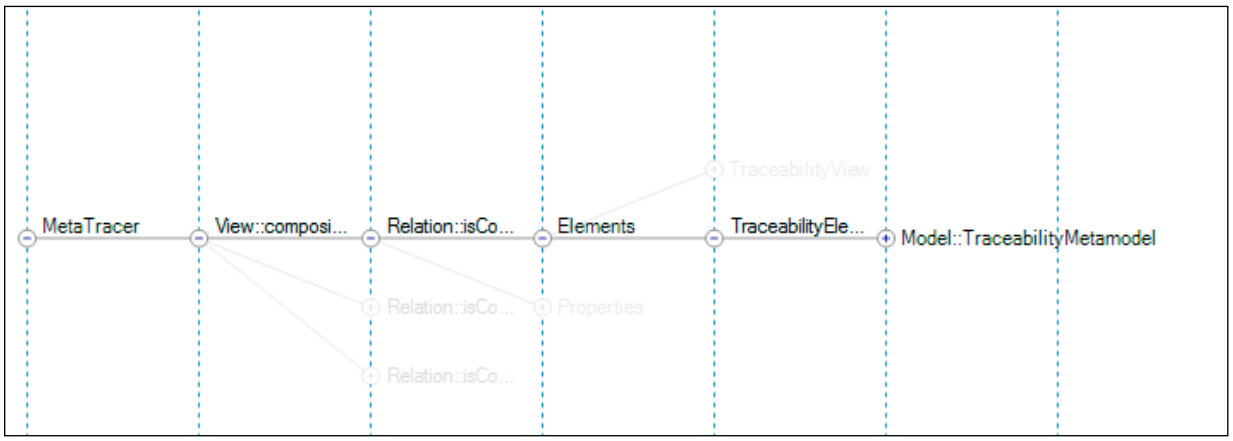
- M2-layer: MOF is divided in two parts, EMOF and CMOF. EMOF, as explained earlier, is not adequate for our purposes, so we use CMOF for its expressiveness and completeness.

- M1-layer: Here we find our proposed Traceability meta-model, as an extension of CMOF. It is upon this meta-model that we construct all our models of traceability.

- M0-layer: this is our runtime traceability visualization.

Traceability visualization

Traceability visualization is still a field that needs to be explored. The visualization that we idealized, tries to tackle the identified problems in traceability models. We could show the models as diagrams and make traceability links between the artifacts related, but it is our opinion that this leads to visual complexity and cluttering. Therefore, we represent our traceability models as navigation-able graph (**Picture 22**).

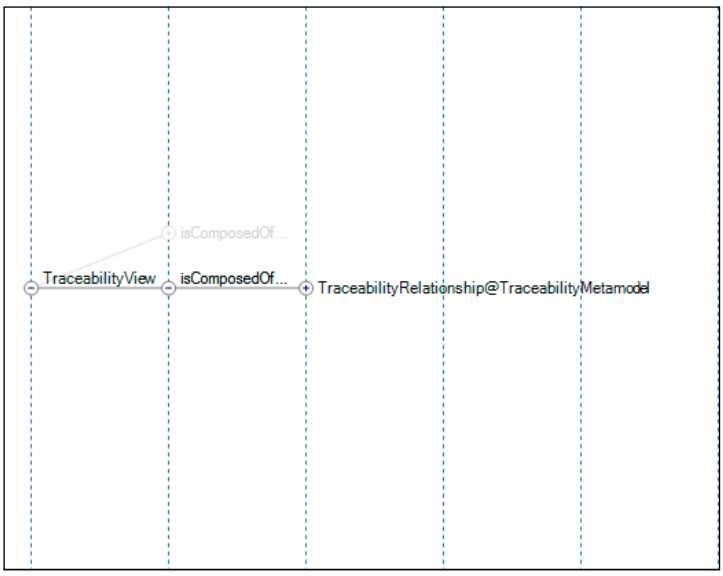


Picture 22: Meta.Tracer traceability visualization

A MetaTracer visualization is a set of TraceabilityView's. By expanding a view, we get the set of TraceabilityRelationships defined. The TraceabilityRelationship are reveled, giving us an overview of the traceability links established between TraceabilityElement. The TraceabilityElement's are gathered in the Elements nodes on the visualization and they represent models or artifacts (traced elements). At the end of each TraceabilityRelationship, we have the set of Properties, which are TraceabilityProperty's defined for the relationship.

Another visualization that we constructed is the change impact visualization. This visualization aims to help the user to visually identify which elements might need changing if we intent to perform any modification on a selected model.

We start by identifying the selected model. We then get all the relationships that specific element is either the source or the target element of the relationship. This is done in all the traceability views, meaning that we get all the relationships from all the views defined by the user. We then get to whom the element is related and identify on which model is the related element (**Picture 23**):



Picture 23: Change impact visualization

Our visualizations might not be the best approach to follow and we are convinced that further testing might reveal its adequacy or not for visualizing traceability information. We expect that future work on this field will contribute for the usefulness of our approach for traceability information.

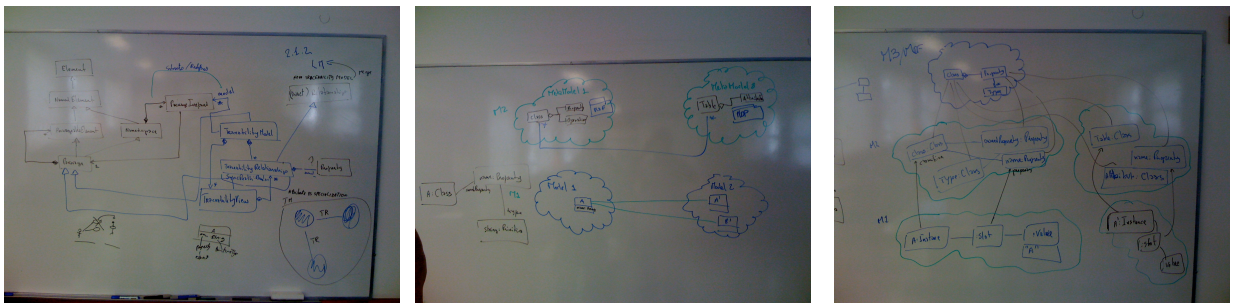
Case study

We wanted to test our traceability meta-model. So we built Meta.Tracer, a tool that allows the user to establish traceability information between already defined models in XMI. Before development effort, we had to consider the limitations and possible problems both on the graphics interface with the user and the goals we wanted to achieve.

Our tool needed to be designed such that the following could be possible:

- Allow the user to import previous models in XMI, built with any of the supported modeling languages, so that we could create traceability information.
- Allow the user to create different perspectives on the traceability information (TraceabilityViews).
- Allow the user to create inter-model relationships and intra-model relationships.
- Allow the user to create his own traceability relationships (TraceabilityRelationships) and semantics (TraceabilityProperties).
- The traceability information would be stored completely separate from the models XMI, to not “pollute” the models with extra information.
- Able to deal with different types of models and artifacts.
- Allow the user to explore the already created traceability information visually.

Working with models, requires us to fully understand how they relate to the MOF architecture (**Picture 24**). We needed to know in which layer did our traceability meta-model belong. We also need to understand that our traceability meta-model is built on MOF, so it can only relate MOF elements. So the related models need to be converted to their MOF representations.

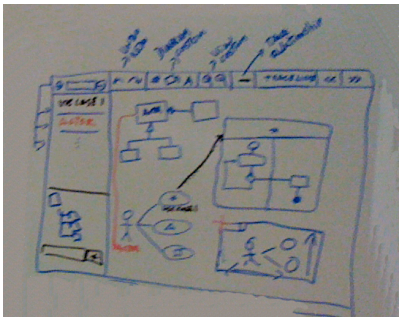


Picture 24: Brainstorming for Meta.Tracer

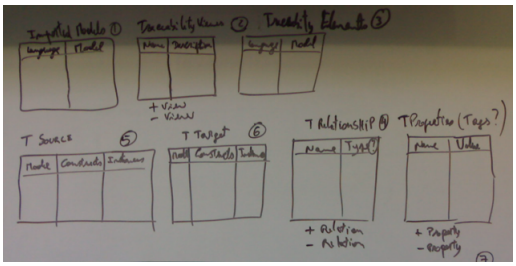
Our first approach was to build a tool that loaded the models visually and then allow the user to create the traceability links between the loaded models. The sketch we came up with (**Picture 25**) made us aware of a problem. The models themselves have a multitude of links (edges), relating the artifacts present on the models. If we added more links between them to depict the traceability information we needed, we would

end up with a cluttered visualization of the models and the traceability information.

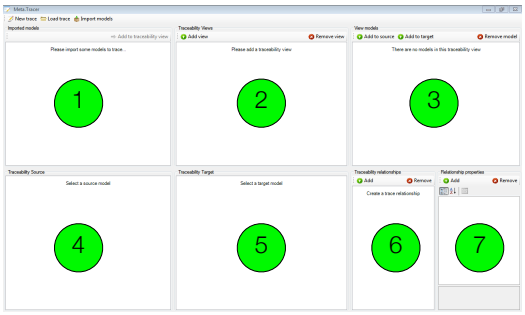
Iterating again on our tool, we focused our efforts on providing firstly a tool for establishing the TraceabilityRelationships between imported models already defined (**Picture 26 and 27**).



Picture 25: Sketch for a visual enabling traceability tool



Picture 26: Brainstorming on the interface that would facilitate the usage of our traceability metamodel.

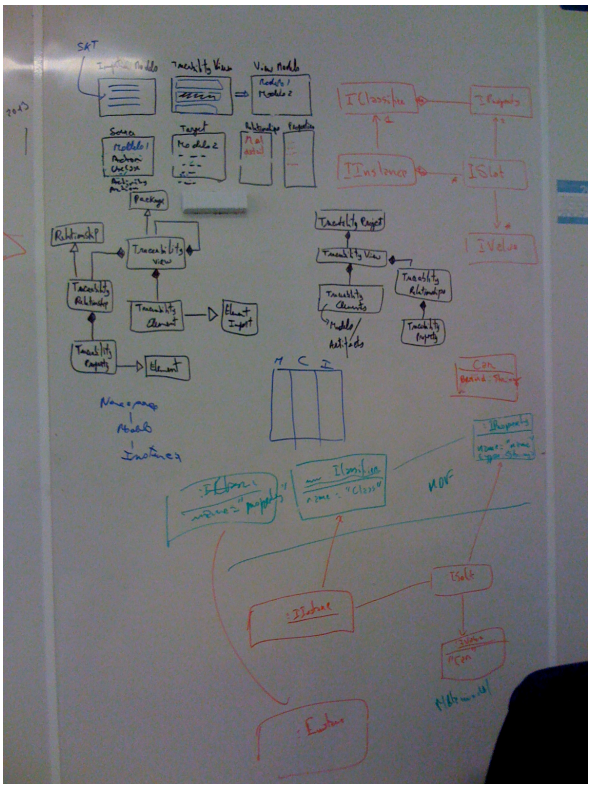


Picture 27: Visual Studio GUI Focusing only on defining traceability information

The user would import models into 1, being these models defined in XMI. After having all the models needed, he would create TraceabilityViews in 2. By double-clicking a row in 1, the model would be associated with the selected TraceabilityView defined in 2. By selecting a row in 2, the user gets a list of all the models related in a TraceabilityView in 3. By using the top toolbar available at 3, the user would then associate a model as the source 4 or target 5 of the relationship he wanted to establish. In 6 the user can create the TraceabilityRelationship and in 7, define the TraceabilityProperties.

Although this interface allows the user to create traceability information between previously defined models, the amount of steps in order to define just one trace, would make this task a tedious one. Another problem is that the user lacked an overview of all the traceability relationships defined.

Focusing on user needs



Picture 28: Rethinking the interface in order to minimize effort by the user.

We wanted to minimize the effort of the user to define traceability information.

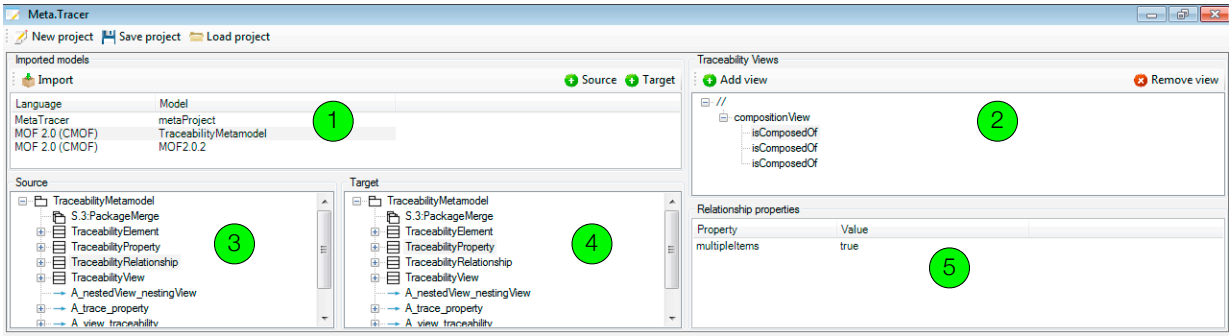
The user would create the views in 2, but in order to view the relationships defined on that view, he had to view them in 6. The user couldn't load more than one model at a time to create traceability relationships in 4 and 5.

So the idea on our forth GUI iteration was to combine 2 and 3 into one interface only. Another priority was to merge 6 and 7 on a single interaction space.

Another problem that we faced was on the space management. This interface occupied by itself the all desktop space. We still needed space for visualization and exploration of the traceability information.

Almost final MetaTracer interface

Creating a traceability model is now done in 5 steps. The user starts by selecting the models he wants to create traceability relationships and adds them to Source and Target containers. He selects the elements he wants to relate on the artifact tree. He then creates the view or perspective upon which he is creating the traceability information he wants to establish. Then, he names the relationship and defines the properties (**Picture 29**).



Picture 29: The final user interface for creating traceability projects

1 Allows the user to import the models previously modeled and stored in XMI. He can also see in which language was the model constructed.

2 Allows the user to create traceability views, define relationships and create traceability properties

3 The user can navigate in all the elements of the model defined as source. He can visually identify and distinguish the different artifacts that are contained in the model. By right-clicking in an element, the user can simulate changes impact, based on the traceability information defined.

4 The user can select another model or even the same model to define intra-traceability relationships. Again, if the user right-clicks an element, he can simulate changes impact.

5 The user can create traceability properties in order to enhance the traceability links semantics. This will allow for the creation of different types of traceability relationships and behaviors.

As mentioned previously, we intended to allow the user to visually explore the traceability information. Since we have managed to reduce the number of containers, and also by made them resizable, the user can now adjust the tool to better fit his needs. We had now space to display a visualization.

So we created two visualizations: a impact change visualization and a traceability overview visualization. Each of these visualizations aim to aid the model-driven software engineer plan and make better modification cost estimates. These two visualizations are just an example of how a traceability meta-model can help towards the model-driven development. Traceability information, by being itself a model, is an abstraction of the connections between models and artifacts.

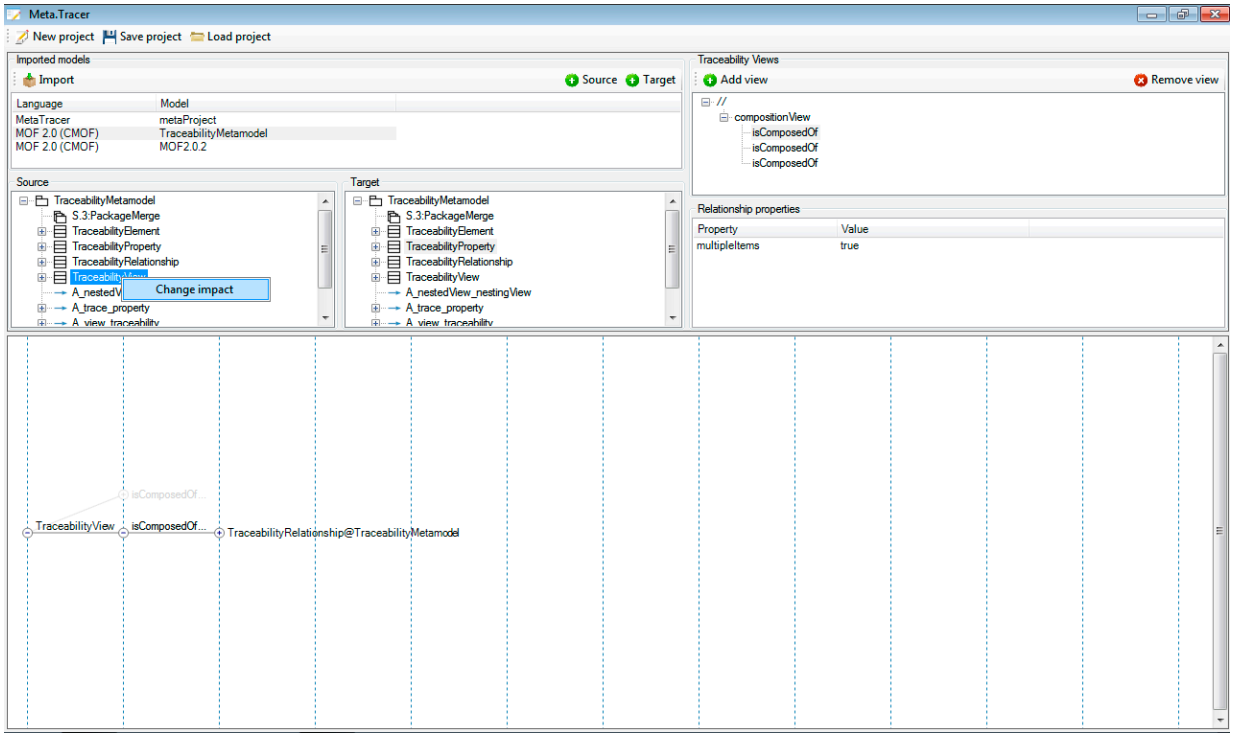
Change impact visualization

The change impact visualization allows the user see with which elements a specific element is related to. By providing this information, a software engineer can evaluate how much of the system needs to be possibly modified if he decides to change a specific model or artifact. He has the information about which relationships are used to relate the elements, in which traceability view it's established, and in which model.

The impact change visualization not only provides information about intra-relationships and also inter-relationships between different models. Information about changing a model could have consequences on other models, and this visualization gives us more control over these situations.

The change impact visualization could be further enhanced, by allowing the user to filter which view did he want to analyze the changes. As we wanted all the traceability information to be available at the same time, we didn't restrict the visualization to a specific view, using the main view for this intent. The main view is the global traceability view, giving us the changes impact in all the relationships and related elements or models.

At this visualization, the select element from the model is the first node. We then get all the relationships where this element is the source or the target of a traceability relationship and in which traceability view was this relationship defined. The related element and model information is visible on the next node.

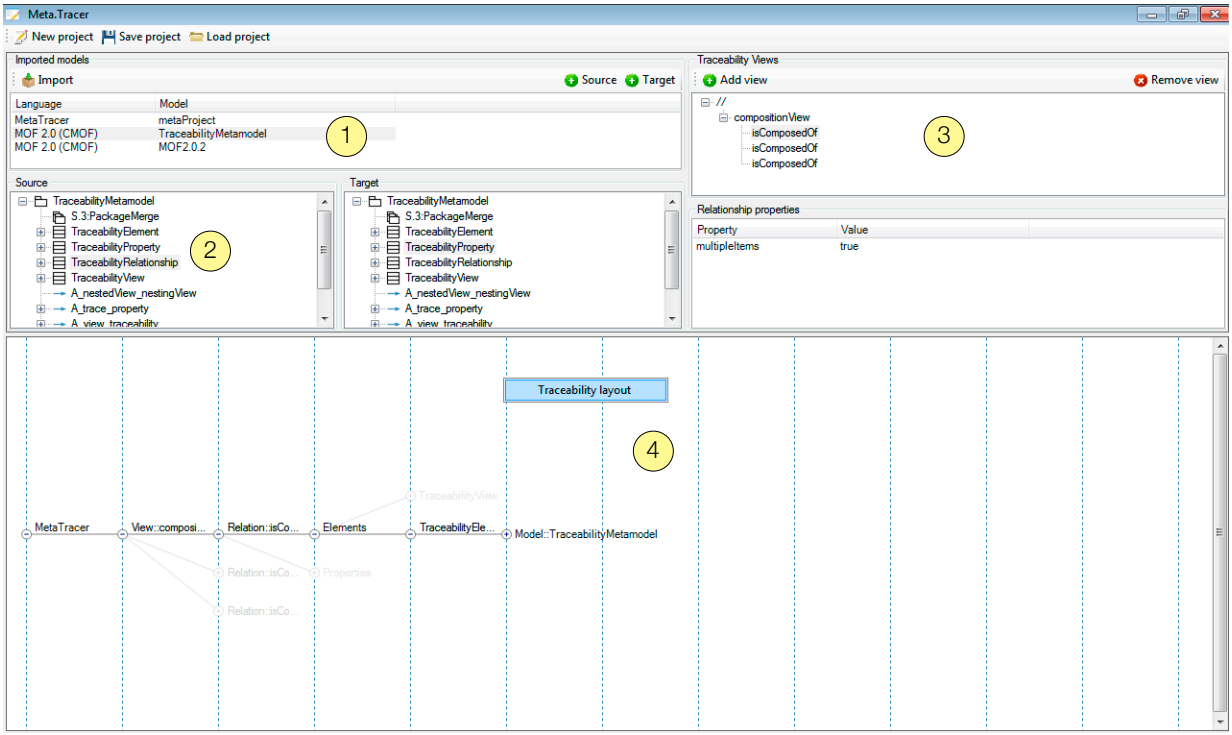


Picture 30: Change impact visualization

Traceability overview visualization

The traceability overview visualization allows the model-driven software engineer to quickly have an better perception on how the models and artifacts are related. Information about which relationships are defined in each TraceabilityView, or which elements are related on a specific perspective or view and which properties were defined for the TraceabilityRelationships that relates them.

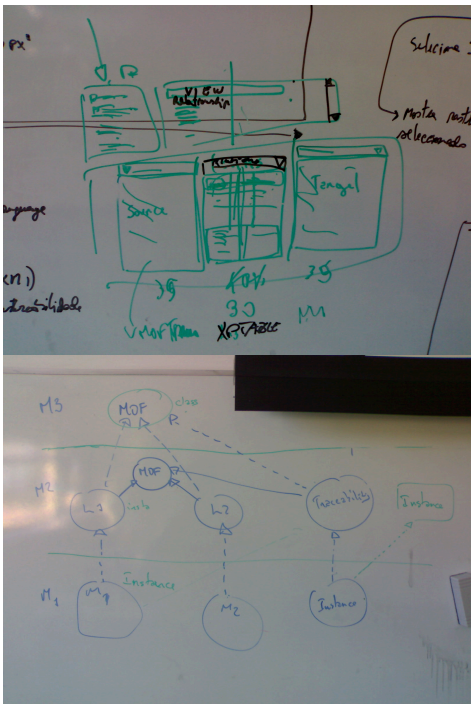
The idea for the traceability overview visualization relies on giving the user an insight on how much traceability information is available to him. The traceability overview visualization starts with MetaTracer. Inside MetaTracer, we have all the defined views. These are the perspectives we want from the traceability information. For example, we can have a Data View, for database traceability relationships. The system is flexible enough for the user to specify his own perspectives. Choosing one of the traceability views, the user gets all the traceability relationships defined in that view. By selecting one of the relationships defined, the user gets the related elements and the properties associated to that relationship. If the user chooses the elements node, he gets the source and target elements and from which model they came from. The properties give us the name and value of the tags defined by the user on the relationship.



Picture 31: Traceability overview visualization

Enhancing user interface - final Meta.Tracer GUI

After using for some time the user interface defined previously, repeating traceability operations over and over again, made us realize that we needed to further enhance the usability of our interface. So we went back to the sketching board (**Picture 32**).



Picture 32: Thinking about enhancing usability

The problems identified with the previous interface were:

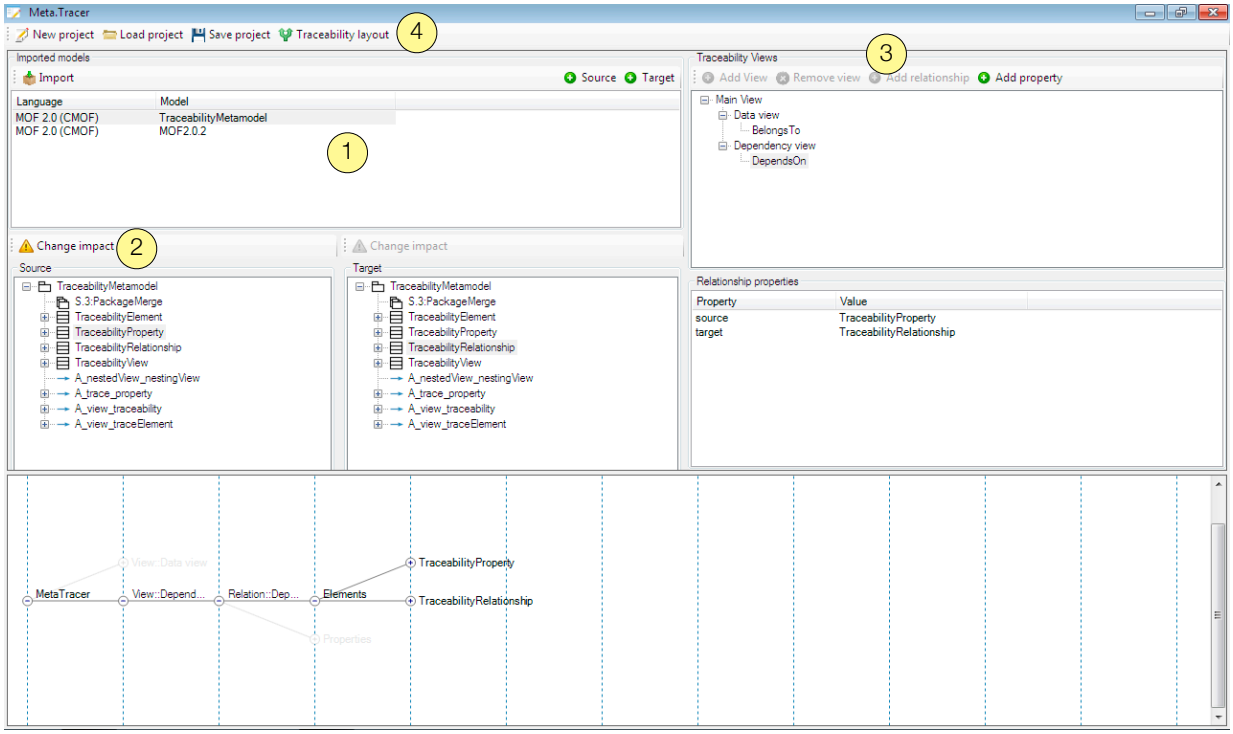
1 Being the project itself saved in XMI, it was being detected as a model available for traceability, so we filtered out the models whose language was MetaTracer.

2 How would the user know that if he right clicked an item in the element tree, would allow him to check for changes impacts? We then added a button above the tree, that activates and deactivates itself regarding wether if there is any traceability information for that element or not.

3 On the menu, the user only had “Add view”. We added “Add relationship” and “Add property” to the menu, so that the user was not required to right click to get a menu option.

4 For making the user aware of the traceability overview visualization, we added a new button “Traceability layout” that does the same as the right click option over the visualization space.

This is the final interface for the user. Further using this interface might reveal possible further usability enhancements (**Picture 33**).



Picture 33: Final Meta.Tracer tool graphical interface

Traceability information exchange - XMI

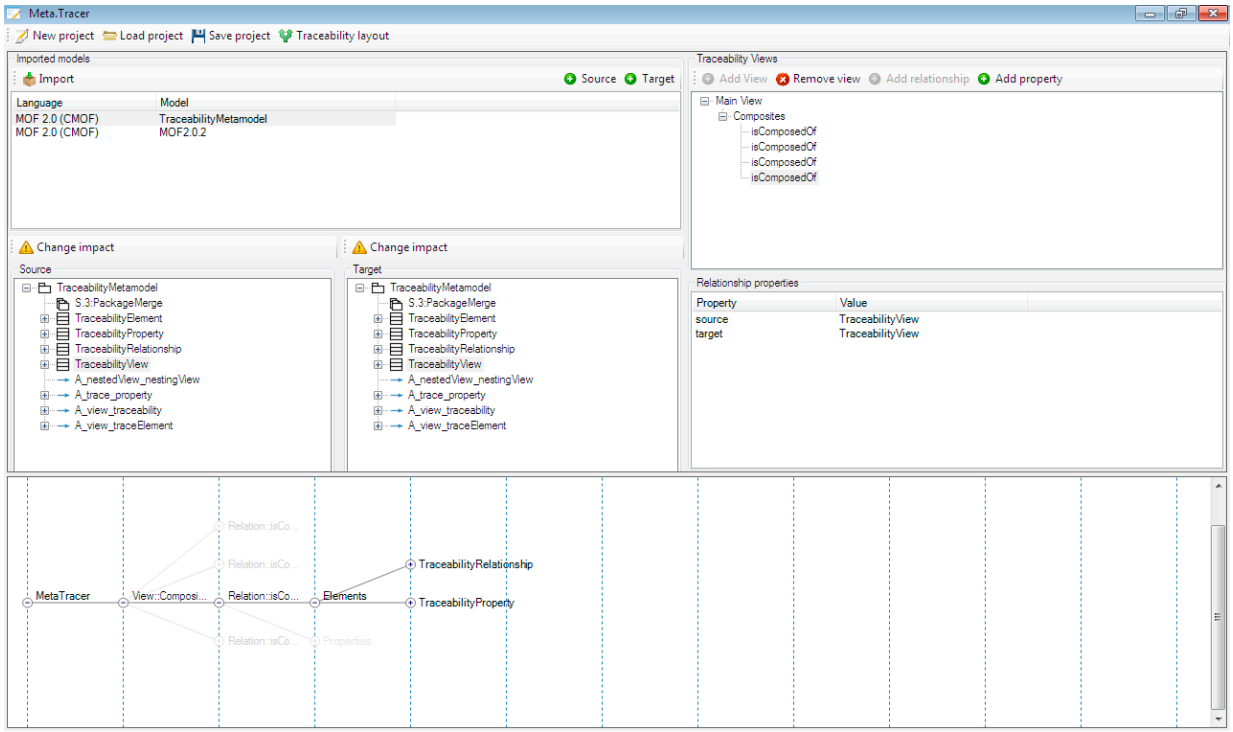
To test the model storage in XMI, we created a traceability project, on which we decided to create the composition perspective of the elements of the traceability meta-model.

We started by loading the traceability meta-model into our tool. Then we set the meta-model model as source and target, since we were creating traceability intra-relationships. We selected the TraceabilityView element on the source, and again the TraceabilityView on the target, because the TraceabilityView can be composed of nested TraceabilityViews.

On the “Traceability views” panel we created our “Composites” view and established the traceability relationship “isComposedOf” between TraceabilityView element from the source and the TraceabilityView element in the target model. Meta.Tracer automatically creates two tags for the user, the “source” and “target” tags, which are as the tag implies, the identities of the elements related. We then created the rest of the “isComposedOf” relationships between TraceabilityView and TraceabilityRelationship, TraceabilityView and TraceabilityElement and finally between TraceabilityRelationship and TraceabilityProperty.

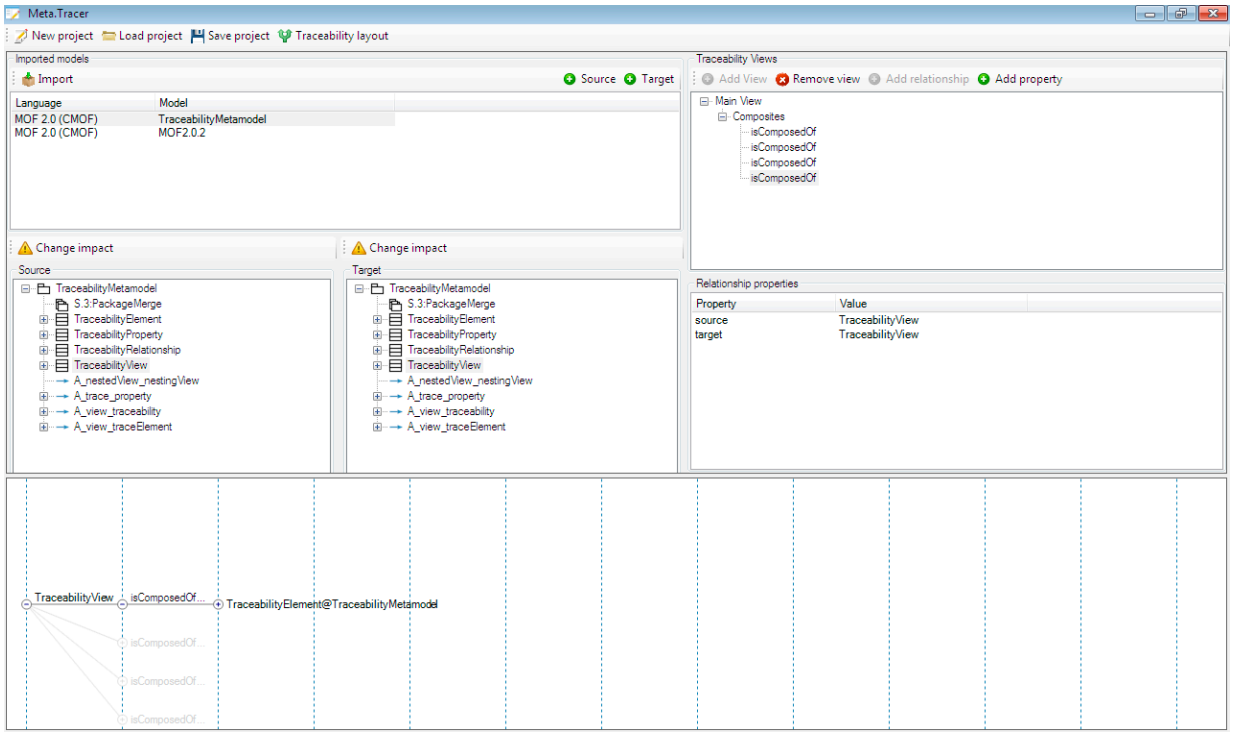
After establishing these relationships, if the user clicks the “Traceability layout” button, Meta.Tracer creates the traceability overview visualization (**Picture 34**).

The interaction with the visualization relies on the user navigation upon the nodes, by clicking on the node he wants to expand or collapse. The nodes expand as long as there is information to display regarding the selected element.



Picture 34: Testing Meta.Tracer by establishing composition traceability information

For testing the change impact visualization, the user has to select an element either on the source or target panel and click the “Change impact” button, or right click on the element and selecting the option accordingly (**Picture 35**).



Picture 35: Testing Meta.Tracer by checking the change impact visualization

Meta.Tracer starts by getting the selected element we want to check change impact. He then gets all the relationships the element is related in. In this particular case, we get the all four relationships that we established before:

- TraceabilityElement: since a TraceabilityView is composed by TraceabilityElements, if we change the TraceabilityView, these elements might need adjusting.

- TraceabilityRelationship: each TraceabilityView is composed by TraceabilityRelationships. So changing the TraceabilityView, might affect TraceabilityRelationship elements.

- TraceabilityProperties: as a TraceabilityView is composed of TraceabilityRelationships, and these in other hand are composed by TraceabilityProperties, changing the TraceabilityView can affect the properties defined for the TraceabilityRelationships.

- TraceabilityView: a TraceabilityView can have nested TraceabilityViews, so by changing the element TraceabilityView, we are affecting all TraceabilityViews.

Saving the project creates an XMI file, that contains all the traceability information, separate from the imported models information, keeping them unaltered. This information can then be used by other tools that follow the same standard and have the definition for the traceability meta-model.

The file saved in this particular example is as follows:


```
<?xml version="1.0" encoding="utf-8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:cmof="http://
schema.labuse.org/spec/MOF/2.0/traceability.xml">
<cmof:TraceabilityView xmi:id="bdeb6ff4-1275-4748-8671-356ed2e9f1be" name="metaProject">
   <nestedTraceView xmi:type="cmof:TraceabilityView" xmi:id="7b75d622-
a24b-4860-8ffc-0c3d472ae6da" name="Composites"
nestingPackage="bdeb6ff4-1275-4748-8671-356ed2e9f1be"
nestingTraceView="bdeb6ff4-1275-4748-8671-356ed2e9f1be">
    <ownedTraceRelationship xmi:type="cmof:TraceabilityRelationship" xmi:id="1f120f42-
fcb3-4d7c-8140-09cebb1bf6a7" info="isComposedOf">
     <ownedProperty xmi:type="cmof:TraceabilityProperty" xmi:id="12c881b9-f777-40a3-ba33-
aee30bc10209" value="TraceabilityView" name="source" />
     <ownedProperty xmi:type="cmof:TraceabilityProperty" xmi:id="5741b50a-9283-4560-
be08-97e7546c59fb" value="TraceabilityElement" name="target" />
    </ownedTraceRelationship>
    <ownedTraceRelationship xmi:type="cmof:TraceabilityRelationship" xmi:id="02c5f138-8b3b-4f17-
b27b-a2dc662532f9" info="isComposedOf">
     <ownedProperty xmi:type="cmof:TraceabilityProperty" xmi:id="4d490b65-e64a-4527-
a6e9-9fee7eccaff9" value="TraceabilityView" name="source" />
     <ownedProperty xmi:type="cmof:TraceabilityProperty" xmi:id="b02edb6b-a7e4-46cd-
b5b0-040bff4f4379" value="TraceabilityRelationship" name="target" />
    </ownedTraceRelationship>
    <ownedTraceRelationship xmi:type="cmof:TraceabilityRelationship" xmi:id="d211fa6f-b016-4305-
a307-3d0848f5e37e" info="isComposedOf">
     <ownedProperty xmi:type="cmof:TraceabilityProperty" xmi:id="6672b8e3-cd9b-4cf9-
af5c-0604a3a46c5c" value="TraceabilityRelationship" name="source" />
     <ownedProperty xmi:type="cmof:TraceabilityProperty"
xmi:id="9c46da6d-9154-4d2b-8621-0cbf3510a265" value="TraceabilityProperty" name="target" />
    </ownedTraceRelationship>
    <ownedTraceRelationship xmi:type="cmof:TraceabilityRelationship"
xmi:id="a20a020a-4eff-428d-9c0d-b7039e855f4e" info="isComposedOf">
     <ownedProperty xmi:type="cmof:TraceabilityProperty"
xmi:id="05745ab4-875c-4f86-8f91-5869520d68a4" value="TraceabilityView" name="source" />
     <ownedProperty xmi:type="cmof:TraceabilityProperty" xmi:id="f17f02a9-84dc-4d30-
a4c8-0a0f6fab6d4f" value="TraceabilityView" name="target" />
    </ownedTraceRelationship>
    <elementImport xmi:type="cmof:TraceabilityElement" xmi:id="5a3bdaa7-
```

c8f2-43b5-963e-7b36871cae9d" importedElement="TraceabilityMetamodel.xmi#S.5"
importingNamespace="7b75d622-a24b-4860-8ffc-0c3d472ae6da" />
<elementImport xmi:type="cmof:TraceabilityElement" xmi:id="b8d11899-b87c-4493-a8eb-
dc778d87cfb5" importedElement="TraceabilityMetamodel.xmi#S.13" importingNamespace="7b75d622-
a24b-4860-8ffc-0c3d472ae6da" />
<elementImport xmi:type="cmof:TraceabilityElement" xmi:id="f8f833bf-ab26-4e4b-97e5-
eeb9797daf88" importedElement="TraceabilityMetamodel.xmi#S.5" importingNamespace="7b75d622-
a24b-4860-8ffc-0c3d472ae6da" />
<elementImport xmi:type="cmof:TraceabilityElement" xmi:id="94e9668c-
c714-425a-99f8-6b60559935a2" importedElement="TraceabilityMetamodel.xmi#S.6"
importingNamespace="7b75d622-a24b-4860-8ffc-0c3d472ae6da" />
<elementImport xmi:type="cmof:TraceabilityElement" xmi:id="1d7d5fee-faff-4c32-a91f-a248317246af"
importedElement="TraceabilityMetamodel.xmi#S.6" importingNamespace="7b75d622-
a24b-4860-8ffc-0c3d472ae6da" />
<elementImport xmi:type="cmof:TraceabilityElement"
xmi:id="befac4f4-4d53-4eff-95c1-13777946be6a" importedElement="TraceabilityMetamodel.xmi#S.18"
importingNamespace="7b75d622-a24b-4860-8ffc-0c3d472ae6da" />
<elementImport xmi:type="cmof:TraceabilityElement" xmi:id="8110f715-c261-49f6-acd4-
b7f8955cf690" importedElement="TraceabilityMetamodel.xmi#S.5" importingNamespace="7b75d622-
a24b-4860-8ffc-0c3d472ae6da" />
<elementImport xmi:type="cmof:TraceabilityElement"
xmi:id="04ea0e90-0742-44fe-9104-3a33c3069b7c" importedElement="TraceabilityMetamodel.xmi#S.5"
importingNamespace="7b75d622-a24b-4860-8ffc-0c3d472ae6da" />
</nestedTraceView>
</cmof:TraceabilityView>
</xmi:XMI>

The file is saved with the UTF-8 encoding, for internationalization support. Then comes the XMI opening tag, that specifies that this particular XMI file is following XMI 2.1 standard and is using the XMI namespace for traceability. The next tag is cmof:TraceabilityView, which is the project main TraceabilityView. As we can see from the rest of the file structure, each xmi element is uniquely identified by the xmi:id tag.

By exchanging the content on this XMI file, it is possible to develop new tools that use this traceability information for new purposes and that was one of our objectives.

Conclusions

When we started this work, our objective in this work was to provide a solution for model-driven development traceability. Our main objectives were to enable traceability information flexible enough to be used in all UML family models and allow the user to explore the information. After researching previous work done in the field of traceability in model-driven development, we soon realized that traceability is much more than a simple connection line between two elements in a model. It has to have a meaning and a purpose.

MOF 2.0 presented us with the core elements we needed to extend UML family with traceability information. Getting the right MOF elements was our first concern after research. We wanted to make sure that we were extending the most adequate elements in order to support traceability. More intensive testing with MOF modeling tools should allows us to test the traceability meta-model. For the traceability view, the Package element was the first to come to mind, as we regard a TraceabilityView as a Package enhanced with traceability information. The TraceabilityElement needed a core element that allowed us to use elements of a model as well as models themselves. A TraceabilityRelationship started as a Relationship, but since we wanted to create directional relationships, as rules can be applied to the relationships depending on the direction of the link, we changed it for the sub-class DirectedRelationship, as it provided us with the source and target concepts. TraceabilityProperties was created as an extension of Element. As mentioned before, by being an Element, it allows us to further enhance the properties with typification. We did not implement property typification, using properties as tags, with name and value attributes.

The advantages of traceability practice are well documented but the manual creation, maintenance and the lack of guidance on what traceability information should be produced is a major drawback for adopting. To this matter, we contributed for a possible way of representing traceability information. To validate our approach, we did a case study, that revealed that our solution is a good contribution for traceability information. We were able to establish relationships between model elements, visualize the traceability information gathered and also analyze change impact on model elements.It was not our intent to provide a tool that would create the traceability models automatically as the user creates new models. If the information we created using the traceability meta-model was created as we generate the artifacts, or at least the skeleton of them, producing the traceability links as we generated the models, we then would only need to fill in the missing details as needed, making traceability part of the process of model-driven development.

As stated by Aizenbud-Reshef N. [5], inconsistencies among models representing different perspectives of a system, or among specifications at different levels of abstraction, can arise during or between the activities of software development, raising the question on how to manage consistency among different models. Although some inconsistencies could be automatically corrected, there are many that cannot, or should not, be automatically corrected. Therefore, applications should instead inform developers of inconsistencies and facilitate the monitoring and resolution of inconsistencies.

As the models are enhanced with traceability information, the user will have knowledge of the impact changes, as they can visualize this information on demand using our tool. But what happens when the user modifies the model on another program? For example, a user deletes an element from the model, that had a traceability relationship with another element, how should we proceed? Our approach is that when the user is loading a traceability project, the relationships that are broken (missing TraceabilityElement) should be highlighted on the traceability overview layout. We then leave the responsibility for changing the

traceability model to the user, so that the information is updated accordingly. We intended to implement this functionality but such was not possible due to time constraints. Consistency checking the traceability models should be more robust than checking if the element exists or not. Deleting a model element should consider model rules and specifications and further work should be done in this matter.

It still remains to be seen if people have an easier time managing a relatively small number of large artifacts with fewer relationships, or if they manage better with a large number of more specialized artifacts, with a correspondingly greater numbers of relationships. The real difficulty of this question becomes obvious when the full life cycle of development is considered. When we add the traceability relationships to the mix, the number of relationships increases almost exponentially, as we can have different traceability perspectives for each model. When we first started Meta.Tracer, we intended to make the tracing like modeling, by drag-and-drop between the models representations. That idea was soon abandoned due to the cluttering that would cause, making it harder to read the models. Our approach was to think of the models as a list of elements, upon which we could create the necessary traceability information. Kolovos D. et al [60] argues that traceability information should be visible on-demand and explores this approach. We look forward for his results to further improve Meta.Tracer.

Whenever a requirement changes or an architectural restriction is introduced, models need to be changed to reflect those decisions. The question needing to be answered is in which model do we reflect the change and the same question is iterated until we made changes in all the affected models. For the project manager, a good traceability system should provide him with the tools to know which model is affected, when it gets changed and provide him with the tools to adjust them accordingly. If the project manager has at his disposal a set of tools designed to help him use model-driven development, since the beginning of the software development cycle, and if those tools support our solution for model traceability and maintains it automatically, traceability would be part of the process for model-driven development. As mentioned before, we were not planning on automating the discovery, creation or maintenance of traceability information and look forward for further work developed in the future upon our contribution.

Using all the strategies proposed by OMG to define a modeling language it enables us to further extend our traceability model in order to incorporate a richer semantic traceability link for anyone's purpose. OCL language can be used for defining restrictions between model artifacts, and these artifacts can be other models themselves. Further development of Meta.Tracer should provide support for QVT, as it not only allows us to define restrictions as it also provides us with the capabilities for model transformations.

Since requirements are often described and specified using natural language, Kurtev I. et al [30] current work in formalizing requirements and change impact analysis provides a better framework for requirement traceability. Unfortunately, traceability is like a puzzle, on which developed work tends to solve an issue at a time. There is the need for facilitating traceability information exchange, upon which various tools could be developed, following the same standards. While we are following OMG standard for specification and information exchange, it is still not sure if XMI standard will prevail and only time will tell as tools are developed that support it.

We look forward towards further work developed in the field of model-driven development, since we believe that it is the future in software development.

References

[1] MOF MetaObject Facility, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF

[2] Nóbrega L., Nunes N. J.,Coelho H.: The Meta Sketch Editor - A Reflexive Modeling Editor, Cpt:17, 201-214, 2008

[3] ISO 12207 - Software lifecycle process standard - http://en.wikipedia.org/wiki/ISO_12207

[4] Hailpern B., Tarr P.: Model-driven development: The good, the bad and the ugly - IBM Systems Journal, vol 45, 2006

[5] Aizenbud-Reshef N., Nolan B.T., Rubin J., Shaham-Gafni Y.: Model traceability - IBM Systems Journal, vol 45, 2006

[6] Brcina R., Riebisch M.: Defining a Traceability Link Semantics for Design Decision Support - ECDMA-TW 2008 : 39-48

[7] UML, Unified Modeling Language, http://www.uml.org/

[8] Davis A.M.: Software requirements: Analysis and Specification - Prentice-Hall, Upper Saddle River, New Jersey (1990)

[9] West M.: Quality Function Deployment in Software Development - IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design 180,5-7 (1991)

[10] Jackson J.: A Keyphrase Based Traceability Scheme - IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design 180, 2-4 (1991)

[11] Pinheiro F.A.C., Goguen J.A.: An Object-Oriented Tool for Tracing Requirements - IEEE Software 13, N.2, 52-64 (1996)

[12] Fowler M.: UML Mode 2003 - http://martinfowler.com/bliki/UmlMode.html

[13] Bézivin J.: On the Unification Power of Models - ATLAS Group (INRIA & LINA) University of Nantes - 2005

[14] Favre J-M.: Megamodelling and Etymology - A story of Words: from MED to MDE via MODEL in five millenniums.

[15] Bastien A., Leblanc H., Coulette B.: A Traceability Engine Dedicated to Model Transformation for Software Engineering - ECDMA-TW 2008 : 7-16

[16] Galvao I., Goknil A.: Survey of traceability approaches in model-driven engineering - EDOC 2007, IEEE Computer Society Press 2007

[17] Eclipse - www.eclipse.org

[18] ATL - ATLAS Transformation Language - http://www.eclipse.org/m2m/atl/

[19] MOF 2.0 Query/View/Transformation language v1.0 - http://www.omg.org/spec/QVT/1.0/

[20] MOF Model to Text Transformation language v1.0 - http://www.omg.org/spec/MOFM2T/1.0/

[21] Kovse J.: Generic Model-to-Model Transformations in MDA: Why and How? - 2002 http://www.softmetaware.com/oopsla2002/kovsej.pdf

[22] IEEE Software Magazine - http://www2.computer.org/portal/web/software

[23] Cao L., Ramesh B., Rossi M.: Are domain-specific models easier to maintain than UML models? - IEEE Software July/August 2009 : 19-21

[24] Sprinkle J., Mernik M., Tolvanen J., Spinellis D.: What Kinds of Nails need a Domain-Specific Hammer? - IEEE Software July/August 2009 : 15-18

[25] Meyer B.: UML - The Positive Spin, 1997 - http://nuyoo.utm.mx/~caff/poo2/UML%20The%20Positive%20Spin.pdf

[26] Unified Modeling Language (Wikipedia) - http://en.wikipedia.org/wiki/Unified_Modeling_Language

[27] Greenfield J.: Microsoft and Domain Specific Languages - http://blogs.msdn.com/jackgr/archive/2004/12/20/327726.aspx (20-12-2004)

[28] Fritzsche M., Johannes J., Zschaler S., Zherebtsov A., Terekhov A.: Application of tracing techniques in Model-driven Performance Engineering - ECDMA-TW 2008 : 111-120

[29] Deursen A., Klint P., Visser J.: Domain-specific languages - http://homepages.cwi.nl/~paulk/publications/Sigplan00.pdf

[30] Backus-Naur Form (BNF) - http://en.wikipedia.org/wiki/Backus–Naur_form

[31] Object Contraint Language - Wikipedia - http://en.wikipedia.org/wiki/Object_Constraint_Language

[32] XMI industry standard ISO/IEC 19503:2005 http://www.iso.org/iso/catalogue_detail.htm?csnumber=32622

[33] Richard F. P., Olsen G. K., Kolovos S.D., Zschaler S., Power C.: Building Model-driven engineering traceability classifications - ECDMA-TW 2008 : 49-58

[34] MDA, Model-driven Architecture, http://www.omg.org/mda/

[35] Object Constraint Language, http://www.omg.org/technology/documents/formal/ocl.htm

[36] Psoda - Visualize Traceability, http://www.psoda.com/cms.php/what-is-psoda/requirements-management/traceability

[37] Rational RequisitePro - IBM Corporation, http://www-306.ibm.com/software/awdtools/reqpro/

[38] Telelogic DOORS, http://www.telelogic.com/products/doorsers/index.cfm

[39] Ramesh B., Jarke M.: Toward reference models for requirements traceability - IEEE Transactions of Software Engineering 27, N.1, 58-93 - January 2001

[40] Pinheiro F., Goguen J.: An Object-oriented tool for tracing requirements - IEEE Software 13, N.2, 52-64 - 1996

[41] Wilcox P., Smith M., Smith A., Pooley R., MacKinnon L., Dewar R.: OPHELIA: An architecture to facilitate software engineering in a distributed environment - Proceedings of the 15th International Conference on Software and Systems Engineering and their applications, Paris, France 2002, Volume 2, 1-7

[42] Hayes J., Dekhtyar A., Osbourne J.: Improving requirements tracing via information retrieval - Proceedings 11th IEEE 2003, 138-150

[43] Marcus A., Maletic J.: Recovering documentation-to-source traceability links using latent semantic indexing - Proceedings 25th IEEE, ACM International conference on Software Engineering 2003, 125-136

[44] Goknil A., Kurtev I., Berg K.: Change Impact Analysis based on Formalization of Trace Relations for Requirements - ECMDA-TW 2008 : 59-76

[45] Sherba S.A., Anderson K.M., Faisal M.: A framework for mapping traceability relationships - Proceedings 2nd International Workshop on Traceability in Emerging Forms of Software Engineering 2003, http://www.soi.city.ac.uk/~gespan/paper5.pdf

[46] Ying A.T.T., Murphy G.C., Ng R., Chu-Carroll M.C.: Predicting source code changes by mining change history - IEEE Transactions on Software Engineering 30, n.9 574-586 2004

[47] Zimmermann T., Weißgerber P., Diehl S., Zeller A.: Mining version histories to guide software changes - Proceedings 26th International Conference on Software Engineering 2004, 563-572

[48] Drivalos N., Paige F.R., Fernandes J. K., Kolovos S. D.: Towards Rigorously Defined Model-to-Model Traceability - ECMDA-TW 2008 : 17-26

[49] Glitia F., Etien A., Dumoulin C.: Fine Grained Traceability for an MDE Approach of Embedded System Conception - ECMDA-TW 2008 : 27-37

[50] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/

[51] Vanhooff B., Baelen S., Joosen W., Berbers Y.: Traceability as input for Model Transformations - ECDMA-TW 07 : 37-46

[52] Anquetil N., Grammel B., Galvão I., Noppen J., Khan S., Arboleda H., Rashid A., Garcia A.: Traceability for Model Driven, Software Product Line Engineering - ECDMA-TW 2008 : 77-86

[53] OMG, Object Management Group, http://www.omg.org/

[54] XMI, XML Metadata Interchange, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#XMI

[55] Bondé L., Boulet P., Dekeyser J.: Traceability and Interoperability at Different Levels of Abstraction in Model Transformations - http://www2.lifl.fr/west/publi/BoBoDe05.pdf (FDL'05 9-2005)

[56] Kolovos D., Paige R., Polack F.: On-Demand Merging of Traceability Links with Models (2006)