Towards Automated Traceability Maintenance in Model Driven Engineering

Mohamed Yassine Haouam and Djamel Meslati

Abstract—**Traceability relations are used to understand the dependencies between the artifacts created during the development of a software system. In model driven Engineering (MDE), traceability relations may be generated implicitly or explicitly. When changes occur to the models, it is necessary that the traceability links must be maintained and must be evolved. The purpose of this paper is to propose an approach for the maintenance of trace links when a transformation was completely or partially invoked.**

In this paper, we have firstly described how traceability links can be stored and how can they be used in an MDE framework. Then we have proposed a traceability maintenance solution based on three main phases: (1) the model comparison phase, (2) the changes detection and classification phase, and (3) the evolution links phase. The proposed approach improves the process of maintaining traceability information in two major ways. First, traces are generated automatically by transformations. This makes the process of establishing traces faster and less error prone compared to manually assigning traces. Second, the (semi-) automated update of traceability relations over time as the software system evolves reduces the manual effort for maintaining traceability relations.

Index Terms—**Model Evolution, Artefacts Co-evolution, Trace Links.**

I. Introduction

DURING a Model-Driven Engineering (MDE) process, the relationships between the artefacts are created both automatically and manually. The relationships between models are often called trace links [1]. To maintain and control these relations, a traceability solution should be a part of every MDE framework [2].

Traceability links are among others used (a) to validate the implementation of requirements, (b) to analyze the impact of changing requirements, and (c) to support regression tests after changes. To ensure all these benefits it is necessary to have a complete and correct set of traceability links between the established artifacts during the software life-cycle. Fully automatic identification of traces without human intervention is impossible. Even in the relatively formal context of model driven software development, establishing and maintaining traceability links is still an issue [3].

In an MDE process, the traceability relations are tightly coupled with software artefacts (models). Trace links are usually stored in the form of tracing models produced by transformations. When changes occur to the artifacts, it is necessary to understand the impact of the development activity on the traceability links and attempts to keep these links in synch with the models. Manual maintenance of traceability can be time-consuming and error-prone due to the large number of potential relationships that exist even for small software systems. In this paper, we provide an approach for the maintenance of trace links when a transformation was completely or partially invoked.

The proposed solution is based on three main phases: (1) the model comparison phase; in which the differences between two model versions are determined, (2) the changes detection and classification phase; that consist of providing information about all the necessary changes to traceability relations, and (3) the evolution links phase; that consist of identifying the updates needed to evolve the trace links.

The remainder of this paper is organized as follows. In Section 2 we describe how traceability links can be stored and used in an MDE framework. The topic of traceability maintenance is discussed in Section 3. Section 4 describes related work in traceability links evolution. Section 5 provides the architecture and process of evolving trace links. An evaluation of the approach is described in Section 6. We end the paper with a summary of related and future work.

II. Traceability in MDE

In software Engineering, traceability often refers to the ability to trace the different stages in the software development process, i.e. trace the evolution of a system from start to finish. The IEEE Standard Glossary of Software Engineering [4] defines traceability as follows:

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

The above definition is strongly influenced by the originators of traceability: the requirements management community [5]. Traceability in requirements engineering is the ability to link project requirements to corresponding design artefacts, resulting software, and associating test cases. However, in MDE a broader definition is required. In [6], traceability is considered as any relationship that exists between artefacts of the software development life-cycle. To support traceability in MDE, it is necessary that the relation between each requirement, its representation in the models and the resulting code sections can be captured, managed, and analyzed [7].

The general consensus amongst the research community is that traceability is mainly used for the following use cases:

Manuscript received November 29, 2015; revised January 27, 2016.

Mohamed Yassine Haouam is an assistant professor in the department of Mathematics and Computer Science at the University of Tebessa, Algeria. He is a member of the Laboratory of Mathematics, Informatics and Systems (LAMIS) and a Ph.D. student at the University of Badji Mokhtar-Annaba (UBMA), Algeria. e-mail: (m.yassine.haouam@gmail.com).

Djamel Maslati is a professor in the department of Computer Science at the University of Badji Mokhtar-Annaba (UBMA), Algeria. e-mail: (meslati.djamel@yahoo.com).

- *Change impact analysis* that provides better understanding of the impact of a change to the cost, schedule and technical aspects of the project.
- *Coverage analysi* that allows validation of which requirements have been fully implemented in the system and which ones are not.
- *Guard against gold platting* which provides a mechanism to make sure that all the features that are present in a system actually correspond to a requirement; as opposed to being unnecessary features that raise the cost and risk of the project.
- *Trade off analysis* is applied when different implementation options exist. Traceability facilitates a trade off analysis by allowing a comparison between the different repercussions of each option. This provides the foundation for further cost-benefit analysis.

A. Storing Traceability Information

Following [8], there are two strategies to storing and managing traceability links. In the first, links are embedded inside the models they refer to in the form of new model elements, while in the second, traceability links are stored in separate models.

1) Intra-model Storage of Traceability Links: Under this approach, traceability links are embedded in the models they refer to in the form of new model elements. Despite its human-friendliness as it represents traceability links as visual model elements that people can easily inspect and navigate, embedding growing amounts of traceability links inside a model causes ”pollution” of the model. We have to bear in mind that traceability is only one of the many concerns when modeling and embedding information relevant to all concerns inside the model can render it overcrowded with elements of secondary importance and can make it difficult to understand and maintain. Another issue is uniformity; in an MDE environment models have their own representations and semantics [9]. Therefore, it is very difficult to distinguish the traceability information from the other model artefacts. As a result, automatic processing of traceability information becomes very challenging.

2) External Storage of Traceability Links: In this strategy, traceability information is stored externally to the models they refer to (i.e. in a separate model). A major advantage of this approach is that source and target models remain clean. In addition, storing traceability links in a model that conforms to a metamodel with clearly defined semantics makes automatic analysis by tools much easier. A requirement of this approach, is that the various model elements have unique identifiers (e.g. xmi.id identifier provided by Meta Object Facility (MOF) and Eclipse Modeling Framework (EMF)), so that the related traceability links can be resolved unambiguously.

B. Generating Traceability Information

Depending on the transformation engine used, the trace links may be generated implicitly or explicitly.

The first approach is often used internally by transformation engines like Queries Views Transformation QVT [10] and Atlas Transformation Language (ATL) [11] to keep track of a transformation [12]. The major advantage of implicit trace link generation is the fact, that no additional effort is necessary to obtain trace links between input and output models, the trace links are generated automatically in parallel to the model transformation. A disadvantage is, that developers have little control of how and when traceability information is created. This means that it is difficult to add additional semantics to the trace links, and means that the traceability information is limited to that defined by the tool.

In [12], the author suggests that with the help of any transformation language supporting two or more output models, like the ATL, it is possible to treat traceability as a regular output model of the transformation and incorporate additional transformation rules to generate it. What he suggests is that in addition to creating mappings from the target model to a source model, there should also be a mapping creating instances of the traceability metamodel, thus creating a traceability model containing information about the transformation. This does however require the existence of a metamodel that describes the external trace model to be created, in the same way as one need a metamodel describing the source and target model of any other transformation. This is a simple solution, and it does not require any additional functionality extending what is already part of the transformation language. One would simply have to create additional ”output patterns” in the mapping rules, that describe the elements of the traceability model to be created. It would however require the developers to create the mapping to a trace model each time they created a mapping. This would make the development process more cumbersome, thus increasing the possibility that someone forgets to do it or does not care to do it, and even more likely; makes errors while doing it.

C. Traceability Maintenance

Providing traceability for a project is not a trivial matter; different activities are necessary to create, maintain and use traceability relations (links).

In [13], Murta et al. characterize the problem of traceability maintenance between architectural elements and source code as follows: ”given an initial set of established traceability links, and given that both an architecture and its implementation can evolve independently, how can traceability links be updated with the addition of new links, removal of existing links, and changes in existing links to ensure that each architectural element is at all times accurately linked to its corresponding source code configuration items, and vice versa?” Without maintenance, traceability relations between elements, i.e. trace links, get lost or represent false dependencies.

Model-based development processes promote the use of models expressed in terms of problem domain concepts (e.g. Entities, Services) as the prime artifact to develop software. These models, to which we refer as high-level models, are used as input for a Model Transformation Chain (MTC). An MTC is a sequence of transformation steps that converts the high-level model, which is rooted in the problem domain, into a low-level model that is rooted in the solution domain (e.g., Java, C#). If this multiphase transformation process is carried out only once, entering each phase only when the preceding phase has been completed, it is a waterfall-like process [14]. Most problems nowadays are too complex to be solved in this manner and mainly development processes are iterative and incremental [15].

A major problem that arises in model-based software development is ensuring that related models evolve consistently while the development proceeds [16]. Maintaining consistency between evolved and affected elements creates many interesting and difficult research problems. Traceability can support this issue by propagating changes that happen to an element in one model to all its related elements in other models [5].

Such changes to referenced model elements can also necessitate maintaining the relationships, i.e. trace links, to reflect all the dependencies between the evolved model elements after the change. Two types of impact can be distinguished:

- The change with no impact on traceability. For example, renaming an attribute in an UML class.
- The change that have impact on traceability. For example, creating a new class.

There are two main approaches for maintaining trace link integrity: event-driven and state based approaches [17]. In the former approach, the elementary changes of the various model elements are constantly monitored and change events are generated based upon these elementary changes. This is achieved by utilizing a set of rules for recognizing the events as constituent parts of intentional development activities. Once these activities have been identified, traceability links related to the changing model elements can be updated automatically. In the second approach, the detection of model changes takes place by comparing different versions of the models and impacted links are found based on the identified changes. If model changes are detected, the trace links, which refer to those models, should be updated. If there are predefined policies associated with the detected changes, then the link maintenance can be done automatically.

III. Related Work

This section describes the related work in the area.

In [18], Cleland-Huang et al. present an approach that can help maintain traceability called event-based traceability (EBT). In this method, requirements and other traceable artefacts of the development process are linked through publish-subscribe relationships. The main components of the system are the requirements manager, the event server, and the subscriber manager. The requirements manager handles the requirements and it is responsible for triggering change events as they occur. The event server manages subscriptions, receives event messages, customizes event notifications according to the process model and subscriptions, and forwards task directives in the form of event notification messages to the subscribers. The subscriber manager is responsible for receiving event notifications and handling them in a manner appropriate to both the artifact being managed and the type of message received. This work discusses a sophisticated change propagation mechanism, enabled by traceability and change recognition (i.e., informing the owner of a related artifact with a detailed message about the changes occur in the requirements specification). The EBT approach presents the event generation, but does not discuss the actual maintenance of impacted traceability relations.

Spanoudakis et al. [19] present a rule-based approach for the automatic generation of traceability relations between documents, which specify either requirement statements or use cases (in structured natural language) and analysis object models. The generation of traceability relations is based on two different types of rules. A first kind of rule, Requirement-to-object-model rules, and a technique based on information retrieval are used to automatically establish traceability relations between requirements and analysis object models. A second kind of rule, inter-requirements traceability rules, is used to trace requirement and use case specification documents to each other. The proposed approach requires the representation of all supported artifacts into the eXtensible Markup Language (XML) format. Due to the use of information retrieval, there is uncertainty within the recognized relations and limited support for developers with false recognition. The approach, in its current form, does not appear to support the maintenance of traceability relations following artifact evolution explicitly, but the approach proposes two interesting ideas. First, the use of extensible and customizable rules that describe properties of expected artifacts in an abstract way. Second, the idea of organizing rules in the style of event, condition, action and to store these rules in the open XML format to facilitate their customization by the user [20].

In [13] [21], the authors describe an approach called ArchTrace. Murta describes how a policy based system of rules can automatically manage traceability links between requirements/architecture to source code. Instead of reconstructing the traceability links after a certain amount of changes or time, ArchTrace updates these links after every commit operation from a user. ArchTrace is only capeable of maintaining existing traceability links, which means that they have to be created manually by the developers or by a traceability recovery method. The ArchTrace tool assume the use of the extensible Architecture Description Language (xADL) for the description of software architectures and the use of Subversion for the versioning of source code. ArchTrace relies on an infrastructure for identifying change events in the architecture models and continuously updating the trace links based on the change events and a set of policies. The policy rule-set has to be configured by the developer to be more accurate in the managing of traceability links. A conflict arises if more than one policy or no policy is triggered by an update. This has to be resolved manually by the developer.

A state-based approach is proposed by Sharif and Maletic [22]. In this approach a difference tool such as EMFCompare is used to identify syntactic differences between different versions of a model. Based on these differences and user input the links are evolved. The authors do not discuss how to update the impacted traceability relations, but the evolution of traceability links by using fine-grained differencing of artifacts that has inspired the approach discussed in this paper.

Inspired by the event generation aspect of the EBT approach, Mder and Gotel [20] present an approach that supports the (semi-)automated update of traceability relations between requirements, analysis and design models of software systems expressed in the UML. It can update traceability relations by analyzing elementary change events that have been captured while working with UML model-

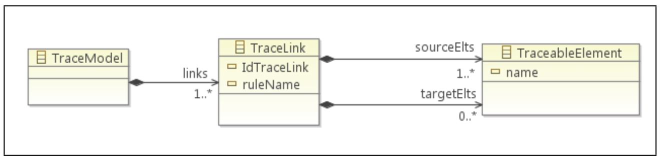


Fig. 1: Traceability Metamodel

ing tool. Within the captured flow of events, development activities comprised of several events are recognized. These are matched with predefined rules gives directive to update impacted traceability relations to restore consistency. While the approach supports the (semi-)automated update of traceability relations, It maintains existing traceability relations whether they are semantically correct or not. It is not possible to find out about the correctness of relations or even to improve their quality via the approach. This means that it requires a reasonable set of initial traceability relations to make the approach useful.

In [23], Uronkarn et al. present an approach that uses the business process change patterns that exist between two versions of a business process model to drive the traceability impact analysis in the presence of business process change. They associate the business process model with traceability information that links the model to other software artifacts, i.e., requirements, use cases, design classes, and programs. When a new version of the business process model is designed to incorporate changes in business requirements, the two versions are compared to determine the types of changes, and identify the affected activities. The developed tool reports the impacted activities in the old version of the BPMN, change patterns that are applied to them, and each kind of software artifacts that are affected.

IV. Overview of the Approach

The approach is concerned with incremental changes to a set of traceability relations. So the maintenance of traceability relations impacted by model evolution. In our approach, trace links are stored in a tracing model. This model represents the relationships between source elements and the target elements generated by the transformation.

A. *Trace Links Generation*

Tracing models must conform to a tracing metamodel. A simple tracing metamodel is presented in figure 1 [12]. This metamodel has two main metaclasses: TraceLink and TracedElement. The first metaclass represents a tracing link that relates a set of source elements with a set of target elements. The second metaclass represents any element that is used by the transformation as a source or any element that is generated by the transformation.

The following example illustrates the trace links generation with a simple example of *simplified bank system* described as UML diagrams. The example is defined using Eclipse Modeling Framework (EMF). The simplified bank system describes a model-to-model transformation that transforms the bank model into a Java model. This transformation is implemented by the Atlas Transformation Language (ATL) [24]. ATL is one of the most widely used transformation languages in the MDE community. It is a hybrid approach language, with both declarative and imperative constructs.

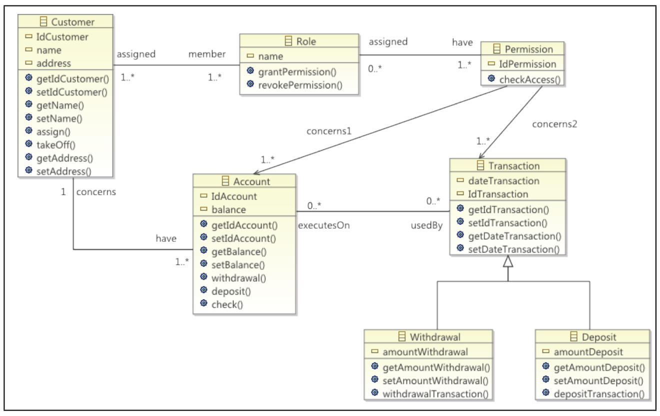


Fig. 2: Bank Model

The source model (Figure 2) is a class diagram of a simplified banking system. The bank model is transformed into a Java model that conforms to a Java metamodel that contains concepts such as Class, Field and Method.

Trace links between input and output model elements are produced by transformation and stored in trace model that is conform to the Trace metamodel presented in figure 1. The approach was introduced in [12].

As an example, Listing 1 shows an ATL transformation rule extended with the tracing generation logic (lines 13-18). An additional target model is specified in transformation header (line 2) to store trace links.

Listing 1: ATL Transformation Rule with Trace Generation

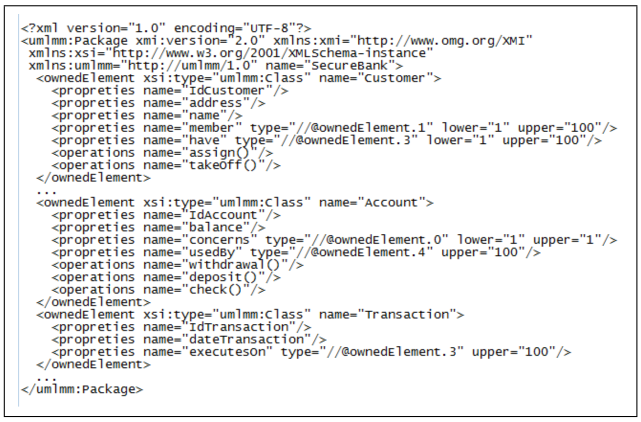
```
 1 module UML2JAVA;
 2 create OUT : JAVA, trace: TRACE from IN : UML;
 3 ...
 4 rule C2C {
 5   from
 6     a:UML!Class
 7   to
 8     b:JAVA!JavaClass(
 9        name<-a.name,
10        isAbstract<-a.isAbstract,
11        superJavaClass<-a.superClass,
12        package<-a.namespace),
13        traceLink:TRACE!TraceLink (
14        ruleName <- 'C2C',
15        targetElts <- Sequence{b})
16        do {
17           traceLink.refSetValue('sourceElts',
                  Sequence {a});
18        }
19 }
```

Figure 3 presents in XMI an excerpt of the input and output models. The dependencies between input elements (Figure 3a) and output elements (Figure 3b) are specified by the trace links in figure 3c. For example, The first trace link (marker 1) specifies that the java class Customer is obtained from the uml class Customer by applying the transformation rule named C2C.

B. *Link Evolution Architecture*

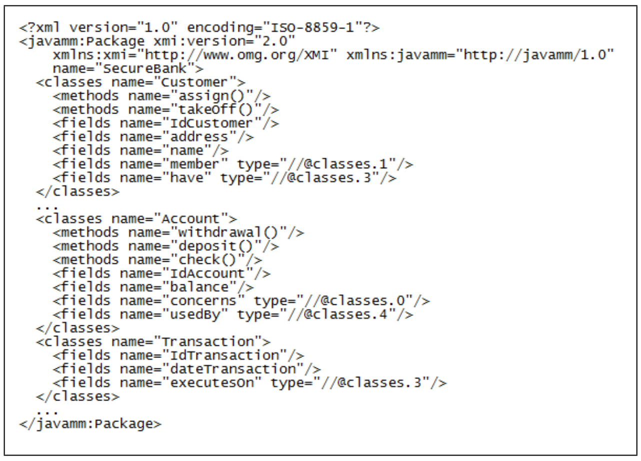
As depicted in figure 4, our work is based on three main phases:

- *Phase 1 (models comparison):* a difference model containing elementary changes was generated by comparing two model versions.



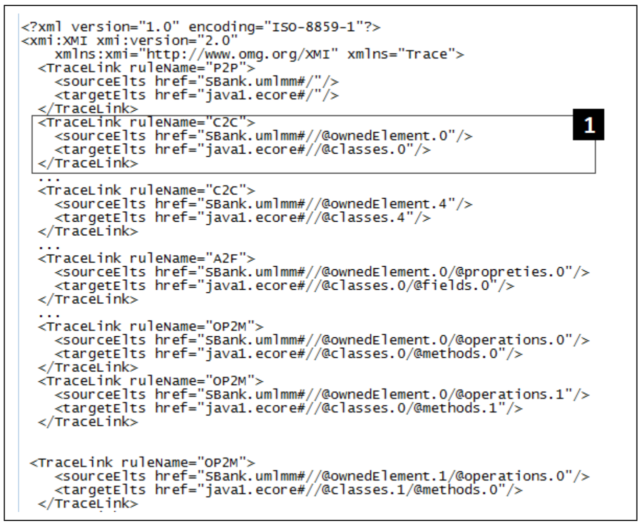
```
<?xml version="1.0" encoding="UTF-8"?>
<umlmm:Package xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:umlmm="http://umlmm/1.0" name="SecureBank">
  <ownedElement xsi:type="umlmm:Class" name="Customer">
    <propreties name="IdCustomer"/>
    <propreties name="address"/>
    <propreties name="name"/>
    <propreties name="member" type="//@ownedElement.1" lower="1" upper="100"/>
    <propreties name="have" type="//@ownedElement.3" lower="1" upper="100"/>
    <operations name="assign()"/>
    <operations name="takeOff()"/>
  </ownedElement>
  ...
  <ownedElement xsi:type="umlmm:Class" name="Account">
    <propreties name="IdAccount"/>
    <propreties name="balance"/>
    <propreties name="concerns" type="//@ownedElement.0" lower="1" upper="1"/>
    <propreties name="usedBy" type="//@ownedElement.4" upper="100"/>
    <operations name="withdrawal()"/>
    <operations name="deposit()"/>
    <operations name="check()"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="Transaction">
    <propreties name="IdTransaction"/>
    <propreties name="dateTransaction"/>
    <propreties name="executesOn" type="//@ownedElement.3" upper="100"/>
  </ownedElement>
  ...
</umlmm:Package>
```

(a) Bank Model



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<javamm:Package xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:javamm="http://javamm/1.0"
    name="SecureBank">
  <classes name="Customer">
    <methods name="assign()"/>
    <methods name="takeOff()"/>
    <fields name="IdCustomer"/>
    <fields name="address"/>
    <fields name="name"/>
    <fields name="member" type="//@classes.1"/>
    <fields name="have" type="//@classes.3"/>
  </classes>
  ...
  <classes name="Account">
    <methods name="withdrawal()"/>
    <methods name="deposit()"/>
    <methods name="check()"/>
    <fields name="IdAccount"/>
    <fields name="balance"/>
    <fields name="concerns" type="//@classes.0"/>
    <fields name="usedBy" type="//@classes.4"/>
  </classes>
  <classes name="Transaction">
    <fields name="IdTransaction"/>
    <fields name="dateTransaction"/>
    <fields name="executesOn" type="//@classes.3"/>
  </classes>
  ...
</javamm:Package>
```

(b) Java Model



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns="Trace">
  <TraceLink ruleName="P2P">
    <sourceElts href="SBank.umlmm#/"/>
    <targetElts href="java1.ecore#/"/>
  </TraceLink>
  <TraceLink ruleName="C2C">                                          1
    <sourceElts href="SBank.umlmm#//@ownedElement.0"/>
    <targetElts href="java1.ecore#//@classes.0"/>
  </TraceLink>
  ...
  <TraceLink ruleName="C2C">
    <sourceElts href="SBank.umlmm#//@ownedElement.4"/>
    <targetElts href="java1.ecore#//@classes.4"/>
  </TraceLink>
  ...
  <TraceLink ruleName="A2F">
    <sourceElts href="SBank.umlmm#//@ownedElement.0/@propreties.0"/>
    <targetElts href="java1.ecore#//@classes.0/@fields.0"/>
  </TraceLink>
  ...
  <TraceLink ruleName="OP2M">
    <sourceElts href="SBank.umlmm#//@ownedElement.0/@operations.0"/>
    <targetElts href="java1.ecore#//@classes.0/@methods.0"/>
  </TraceLink>
  <TraceLink ruleName="OP2M">
    <sourceElts href="SBank.umlmm#//@ownedElement.0/@operations.1"/>
    <targetElts href="java1.ecore#//@classes.0/@methods.1"/>
  </TraceLink>


 <TraceLink ruleName="OP2M">
    <sourceElts href="SBank.umlmm#//@ownedElement.1/@operations.0"/>
    <targetElts href="java1.ecore#//@classes.1/@methods.0"/>
  </TraceLink>
```

(c) Trace Model

Fig. 3: Trace Links Generation

- *Phase 2 (Changes detection and classification)* consists of providing information about all the necessary changes to traceability relations.
- *Phase 3 (Evolve links)* updates the trace links related to the changed model elements.

1) Model Comparison: In the first step, the original and evolved models was compared. All differences between the two versions are stored in a difference model [25]. This difference model can contain all kinds of changes of Ecore

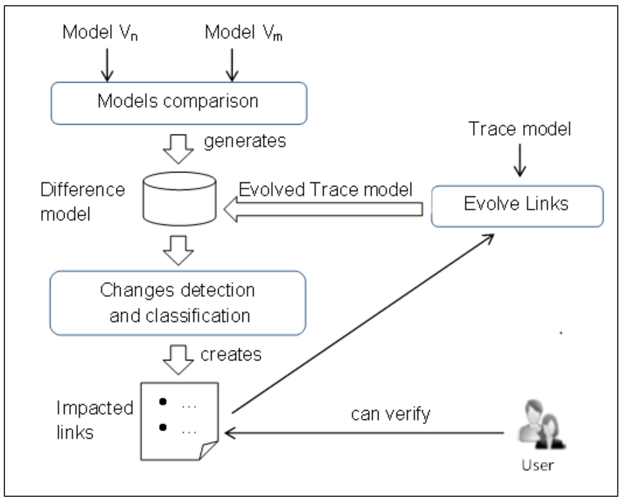


Fig. 4: Trace Links Evolution Architecture

elements such as additions, deletions, renaming or movements of model elements or other updates of model element properties such as the cardinality of an attribute. In our approach, we consider three fundamental change types:

- *Adding a new element (Add):* This change enhances a model by adding a new element. The new model element can be used by transformations to generate new artifacts and may require the creation of new traceability relations that will be added to the trace model.
  Required traceability update: After adding a new element, it might be necessary to re-execute the transformations in order to generate the corresponding elements of the associated models. The trace links generated implicitly or explicitly by transformations specifies what elements are linked to the new element, so it is possible to evolve the trace model by creating one or more traceability relations on the new element.
  Impact on existing traceability: This change type has no impact on the existing traceability relations but, new relations will be added to the existing relations within the trace model to ensure the completeness of trace links.
- *Deleting an element (Delete):* An element may be removed from a model. While removing an element, its relations will also be deleted, so trace links from the trace model are removed.
  Required traceability update: If an element has been removed it is necessary to remove its traceability relations from the trace model.
  Impact on existing traceability: The deletion of traceability relations has no impact on relations to independent elements. If target elements are linked to other elements, it is necessary to check whether these elements are still valid and required or not.
- *Modifing an Element (modify):* The modification may affect the names of elements (e.g., renaming a package, a class, an attribute, an association or an operation) or the values of their properties (e.g., the lower bound of an association).
  Required traceability update: If an element has been modified it is necessary to maintain all traceability relations of the renamed model element on the trace

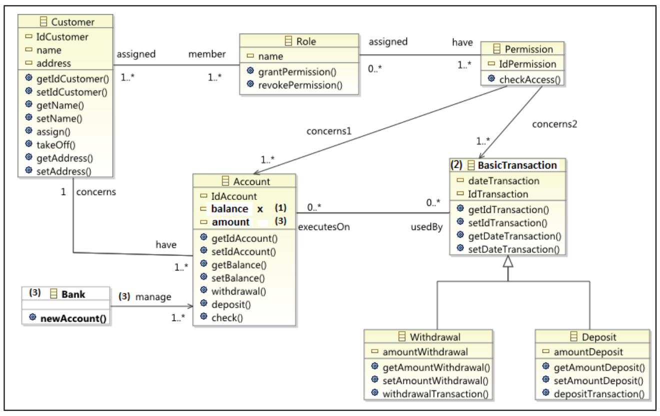


Fig. 5: Evolved Version of the UML Class Diagram for the Bank System

model.

Impact on existing traceability: This change type has no impact on the existing traceability relations.

Figure 5 presents an evolved version of the UML class diagram for the bank system described in the previous section. The modifications introduced are (1) the deletion of an attribute (*balance*), (2) the modification of *Transaction* class name by *BasicTransaction*, (3) the introduction of a new class *(Bank)*, a new method *(newAccount)*, a new association *(manage)* connects the *Bank* and the *Account* classes, and a new attribute *(amount)* is added to the *Account* class.

The computation of the differences between model versions is performed with The Eclipse plug-in EMF Compare [26]. This tool provides algorithms to calculate the delta between two versions of a model and visualizes them using tree representations. In Figure 6 it is depicted the computed difference model between two versions of the bank model.

The computing of differences between the two model versions is illustrated in listing 2. The comparison process is divided in 2 phases: *matching* and *differencing*. In the matching phase the elements that belong together are determined (line 21). The differencing step (line 23) creates the list of changes based on the input models and the previously created match.

Listing 2: Model Comparison with EMF Compare

```java
1  import org.eclipse.emf.compare.diff.metamodel.
       DiffElement;
2  import org.eclipse.emf.compare.diff.metamodel.
       DiffModel;
3  import org.eclipse.emf.compare.diff.service.
       DiffService;
4  import org.eclipse.emf.compare.match.metamodel.
       MatchModel;
5  import org.eclipse.emf.compare.match.service.
       MatchService;
6  import org.eclipse.emf.compare.util.ModelUtils;
7  import org.eclipse.emf.ecore.EObject;
8  import org.eclipse.emf.ecore.resource.ResourceSet;
9  import org.eclipse.emf.ecore.resource.impl.
       ResourceSetImpl;
10 public class compare {
11 public static void main (String[] args) throws
       IOException,
12 InterruptedException {
13 // Load the two input models
14 ResourceSet resourceSet1=new ResourceSetImpl();
15 ResourceSet resourceSet2=new ResourceSetImpl();
16 URI uri1=URI.createFileURI("model/BankV1.ecore");
17 URI uri2=URI.createFileURI("model/BankV0.ecore");
18 EObject model1=ModelUtils.load(uri1,resourceSet1);
19 EObject model2=ModelUtils.load(uri2,resourceSet2);
20 // Matching
21 MatchModel match=MatchService.doMatch(model1,model2,
       Collections<String,Object>emptyMap());
22 // Compare the two models
23 DiffModel diff=DiffService.doDiff(match,false);
24 ModelUtils.save (diff ,"model/diffxmi");
25 // List of differences
26 List <DiffElement>differences=diff.getDifferences();
27 for ( DiffElement diffElement:differences ) {
28     System.out.println("Change type : "+ diffElement.
           getKind());
29     System.out.println(diffElement.toString()); }
30 }
31 }
```

```
Change type: Change
Attribute name in BasicTransaction has changed from Transaction to
BasicTransaction

Change type: Addition
org.eclipse.emf.ecore.impl.EClassImpl@e49d67c (name: Bank)
(instanceClassName: null) (abstract: false, interface: false) has been added

Change type: Addition
org.eclipse.emf.ecore.impl.EAttributeImpl@97494c8 (name: amount) (ordered:
true, unique: true, lowerBound: 0, upperBound: 1) (changeable: true,
volatile: false, transient: false, defaultValueLiteral: null, unsettable:
false, derived: false) (iD: false) has been added

Change type: Deletion
org.eclipse.emf.ecore.impl.EAttributeImpl@203fa5ac (name: balance) (ordered:
true, unique: true, lowerBound: 0, upperBound: 1) (changeable: true,
volatile: false, transient: false, defaultValueLiteral: null, unsettable:
false, derived: false) (iD: false) has been removed
```

Fig. 7: Differences between the Compared Models

Figure 7 illustrates the results of applying the comparison process to our case study. In this figure it is possible to see the following changes:

- The element *Transaction* is renamed.
- The elements *Bank* and *amount* are added
- The element *balance* is deleted.

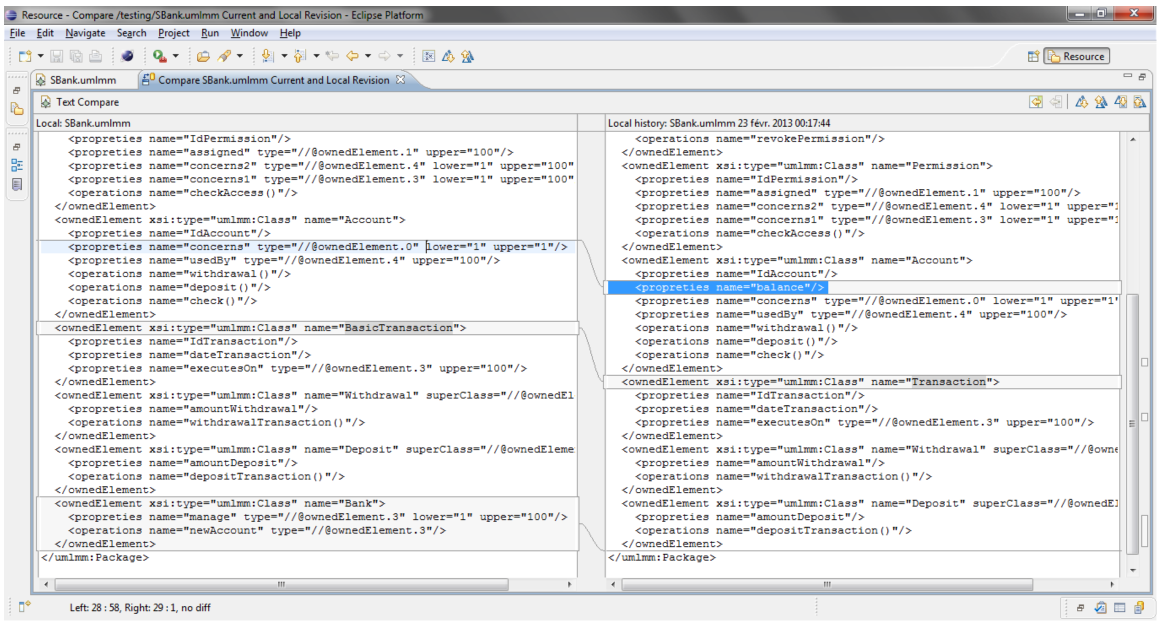
2) Changes detection and classification: Once changes have been properly detected and represented, they are added to an update list which eventually provides information about all the necessary changes to traceability relations. These changes have to be classified into one of two categories:

- *Changes that require traceability updates*. In our case, the changes (1) the deletion of element "balance" and (3) the addition of elements "Bank", "newAccount", "manage", and "amount" are classified in this category.
- *Changes without impact on existing traceability relations*. For example, the change (2) the modification of element "Transaction" is classified in this category.

3) Evolve links: The changes obtained as results of the previous phase are used to identify the updates needed to evolve the trace links after model evolution. Correlating to the changes types discussed in the subsection above, the following changes and associated traceability updates to the trace model are possible:

- The update action for relations on the update source(s) is defined to stay if the source element still exists.
- If the element has been removed during the evolution activity, all trace links connecting any element with the deleted item should be removed.
- If new source element is added, new relations in the trace model are defined to be created.
- All existing relations of all modified elements are defined to stay. These relations are not impacted by the evolution activity.

The basic structure of the evolution rule is illustrated in listing 3. Each evolution rule has a name (line 1) and a


Resource - Compare /testing/SBank.umlmm Current and Local Revision - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Resource
SBank.umlmm
Compare SBank.umlmm Current and Local Revision
Text Compare

Local: SBank.umlmm

```
    <propreties name="IdPermission"/>
    <propreties name="assigned" type="//@ownedElement.1" upper="100"/>
    <propreties name="concerns2" type="//@ownedElement.4" lower="1" upper="100"
    <propreties name="concerns1" type="//@ownedElement.3" lower="1" upper="100"
    <operations name="checkAccess()"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="Account">
    <propreties name="IdAccount"/>
    <propreties name="concerns" type="//@ownedElement.0" lower="1" upper="1"/>
    <propreties name="usedBy" type="//@ownedElement.4" upper="100"/>
    <operations name="withdrawal()"/>
    <operations name="deposit()"/>
    <operations name="check()"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="BasicTransaction">
    <propreties name="IdTransaction"/>
    <propreties name="dateTransaction"/>
    <propreties name="executesOn" type="//@ownedElement.3" upper="100"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="Withdrawal" superClass="//@ownedEl
    <propreties name="amountWithdrawal"/>
    <operations name="withdrawalTransaction()"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="Deposit" superClass="//@ownedEleme
    <propreties name="amountDeposit"/>
    <operations name="depositTransaction()"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="Bank">
    <propreties name="manage" type="//@ownedElement.3" lower="1" upper="100"/>
    <operations name="newAccount" type="//@ownedElement.3"/>
  </ownedElement>
</umlmm:Package>
```

Local history: SBank.umlmm 23 févr. 2013 00:17:44

```
    <operations name="revokePermission"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="Permission">
    <propreties name="IdPermission"/>
    <propreties name="assigned" type="//@ownedElement.1" upper="100"/>
    <propreties name="concerns2" type="//@ownedElement.4" lower="1" upper="1
    <propreties name="concerns1" type="//@ownedElement.3" lower="1" upper="1
    <operations name="checkAccess()"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="Account">
    <propreties name="IdAccount"/>
    <propreties name="balance"/>
    <propreties name="concerns" type="//@ownedElement.0" lower="1" upper="1'
    <propreties name="usedBy" type="//@ownedElement.4" upper="100"/>
    <operations name="withdrawal()"/>
    <operations name="deposit()"/>
    <operations name="check()"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="Transaction">
    <propreties name="IdTransaction"/>
    <propreties name="dateTransaction"/>
    <propreties name="executesOn" type="//@ownedElement.3" upper="100"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="Withdrawal" superClass="//@owne
    <propreties name="amountWithdrawal"/>
    <operations name="withdrawalTransaction()"/>
  </ownedElement>
  <ownedElement xsi:type="umlmm:Class" name="Deposit" superClass="//@ownedEl
    <propreties name="amountDeposit"/>
    <operations name="depositTransaction()"/>
  </ownedElement>
</umlmm:Package>
```

Left: 28 : 58, Right: 29 : 1, no diff

Fig. 6: Difference Model

description (line 2) revealing what the evolution rule is used for. (line 4) specifies which conditions have to be fulfilled by a model difference to lead to the execution of the action part. The condition part is a boolean expression. In each condition part, multiple conditions can be combined such as boolean expressions can be combined in Java using logical operators. For each type of model change which can be found in the model comparison activity , we offer one predefined condition, which checks whether the particular type of model change occurred. For example, the condition addAssociation indicates, whether the currently considered difference model represents the addition of an association.

Listing 3: Basic Structure of an Impact Rule

```
1 evolveRule "<name>" {
2    description = "..."
3    evolve {
4       (<condition part> => <action part>)
5  }
6 }
```

V. Evaluation

A. Objective and Research Questions

The objective of our case study is to validate the applicability of our approach with respect to the manual effort that can be saved and quality that can be reached with the maintained traceability while implementing changes to a software development project. For this purpose, we derived the following research questions.

- *Research question 1: manual effort:* Does the use of the approach reduce the manual effort necessary for generating and maintaining traceability relations? The manual effort for traceability maintenance refers to the time the developer spends on this task. It comprises navigating within the evolved models and performing the required changes to update the impacted traceability relations.
- *Research question 2: maintenance quality:* Are the qualities of traceability relations performed by the approach comparable or better to those maintained manually? To determine the quality of a set of traceability relations depends upon having an agreed baseline. Three types of changes to the traceability relations are then distinguished:
  $\triangle c$ Changes that have been performed correctly according to the baseline.
  $\triangle i$ Changes that have been performed incorrectly.
  $\triangle m$ Changes that have not been performed (missing changes).
  To be able to compare the number and the quality of the changes we compute two measures that are commonly used to evaluate approaches dealing with uncertainty in recognition processes, precision and recall. Precision relates correct changes to all performed changes and tells us about the quality of performed changes:
  $$QP = \triangle c/(\triangle c + \triangle i)$$
  Recall relates correct changes to all required changes and tells us about the number of necessary changes performed:
  $$QR = \triangle c/(\triangle c + \triangle m).$$

B. Design and Subject of the Study

We decided to apply our approach on the example presented in section IV-A. The system artifacts included models on two levels of abstraction: class diagram and java diagram. The class diagram contained 7 classes, 11 attributes, and 28 methods. The initial set of trace links was generated by transformation and stored in a trace model that is conforming to a trace metamodel. The set of traceability relations obtained for this software system contained 47 trace links.

In order to answer the first research question, we measured the execution times of applying the approach on the UML class diagram, as described before. On that one, we performed 23 change operations, covering all types of UML class diagram changes. Based on a pilot study with two

TABLE I: Traceability Links Evolution

	Correctly performed	Incorrectly performed	Lost	Total
Traceability Links	48	8	4	60
%	80%	13%	7%	100%

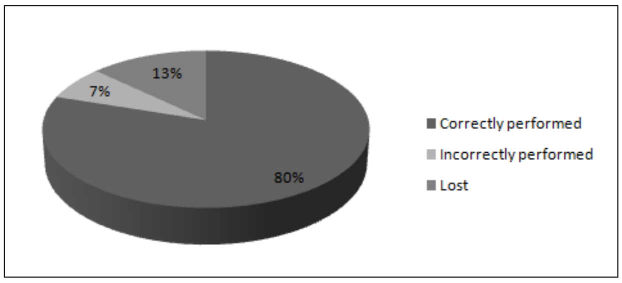


Fig. 8: Quantitative Analysis

master students, it was estimated that it would take 1-2 hours to manually maintain the traceability relations.

To answer the second research question, 60 UML class diagram elements were added, renamed, or removed. The set of traceability links evolved by our approach are compared with the set of ideal traceability links detected manually.

C. Results and Discussion

The first phase of the study showed that the complete time for performing trace links evolution by using our approach was 40 minutes on average. This time includes links generation, model comparisons, and links evolution. The undetected changes or those detected with ambiguity require a decision from the user. Taking into account the time that the user spends to resolve the changes untreated by the approach, the difference is statistically significant.

In the second phase of the study, the trace model evolution analysis allowed us to find out correct, incorrect and missing changes to the set of traceability relations. A summary of the obtained results is shown in Table I.

Figure 8 presents the results of the analyses, illustrating that, our approach correctly performed 80% of traceability links and 13% of links were incorrectly performed. Moreover, 7% traceability links were lost. These links were lost when elements were created with the same properties of those that had been removed. This sequence of change operations usually generates a ”rename” change type, instead of the two change types, ”delete” and ”add”.

To put these figures in perspective, we borrow two metrics from the information retrieval field: precision and recall. The computed precision QP and recall QR provide information about the correctness and completeness of changes to the link set. These two metrics apply here in the sense that we can use precision to show the percentage of actually identified traceability links that are correct ($QP = 86\%$; showing that 86% of changes have been correctly detected and performed by the approach and 14% of the traceability links that were found are inaccurate) and recall to show the percentage of ideal traceability links that were actually identified ($QR = 92\%$; showing that we missed merely 8% of the traceability links that should have been found).

VI. Conclusion and Future Work

This paper has presented a new approach for managing the evolution of traceability links when a model-to-model transformation was completely or partially invoked. The proposed approach is based on three phases. (1) A difference model containing elementary changes was generated by comparing two model versions; (2) The changes detected in the previous phase are added to an update list which eventually provides information about all the necessary changes to traceability relations. (3) Once the model changes are identified, a set of updating operations must be performed to maintain the traceability relations.

Our approach improves the process of maintaining traceability information in two major ways. First, traces are generated automatically by transformations at a very low cost. Second, our approach has to detect each model change operation and automatically update traceability relations.

While our approach significantly improves the evolution of traceability links, our work to date also highlights that further work remains to be done. First and foremost, we recognize that achieving 100% precision and 100% recall is the ultimate result to be achieved by our approach. This, however, may or may not be unrealistic. Secondly, we plan to introduce other types of model changes.

References

[1] G. K. Olsen and j. Oldevic, “Scenarios of traceability in model to text transformations,” in *The ECMDA-FA 2007. LNCS, Springer-Verlag*, 2007, pp. 144–156.

[2] S. Walderhaug, U. Johansen, E. Stav, and J. Aagedal, “Towards a generic solution for traceability in mdd,” in *European Conference on Model-Driven Architecture Traceability Workshop (ECMDA-TW06), Bilbao, Spain 2006*, pp. 41–50.

[3] S. Winkler and J. Pilgrim, “A survey of traceability in requirements engineering and model-driven development,” in *Software and Systems Modeling (SoSyM)*, 2010, pp. 529–565.

[4] IEEE, “IEEE standard glossary of software engineering terminology,” *IEEE Std 610.12-1990 , New York*, 1990.

[5] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni, “Model traceability,” *IBM Systems Journal - Model-driven software development*, vol. 45, no. 3, pp. 515–526, 2006.

[6] N. Drivalos, R. F. Paige, K. Fernandes, and D. S. Kolovos, “Towards rigorously defined model-to-model traceability,” in *The ECMDA Traceability Workshop, Berlin, Germany 2008*, pp. 17–26.

[7] M. Aleksy, T. Hildenbrand, C. Obergfell, and M. Schwind, “A pragmatic approach to traceability in model-driven development,” in *The PRIMIUM - Process Innovation for Enterprise Software 2009.*

[8] D. S. Kolovos, R. F. Paige, and F. A. Polack, “On-demand merging of traceability links with models,” in *ECMDA-TW 2006 Proceedings, Bilbao, Spain 2006*, pp. 7–15.

[9] R. Brcina and M. Riebisch, “Defining a traceability link semantics for design decision support,” in *The ECMDA-TW, Berlin, Germany 2008.*

[10] OMG, “OMG paper (2008),” *Retrieved January 2014, http://www.omg.org/spec/QVT/1.0*, 2008.

[11] F. Jouault and I. Kurtev, “Transforming models with ATL,” in *The ECMDA Workshop on Traceability, Germany 2005.*

[12] F. Jouault, “Loosely coupled traceability for ATL,” in *The Model Transformations in Practice Workshop at MoDELS, Montego Bay, Jamaica 2005.*

[13] L. G. P. Murta, A. van der Hoek, and C. M. L. Werner, “Archtrace: policy-based support for managing evolving architecture-to-implementation traceability links,” in *The 21st IEEE/ACM International Conference on Automated Software Engineering, Tokyo, Japan 2006*, pp. 135–144.

[14] W. W. Royce, “Managing the development of large software systems: concepts and techniques,” in *The 9th international conference on Software Engineering ICSE87, IEEE Computer Society Press. Los Alamitos, CA, USA 1987*, pp. 328–338.

[15] I. Jacobson, J. Rumbaugh, and G. Booch, *The unified software development process*, object technology series ed. Addison-Wesley, 1999.

[16] Z. Huzar, L. Kuzniarz, G. Reggio, and J. L. Sourrouille, "Consistency problems in uml-based software development," in *Nunes, N.J., Selic, B., da Silva, A.R., lvarez, J.A.T. (eds.), UML Satellite Activities, Lecture Notes in Computer Science 3297, Springer 2004*, pp. 1–12.
[17] N. Drivalos, D. S. Kolovos, and R. F. Paige, "A state-based approach to traceability maintenance," in *The 6th ECMFA Traceability Workshop (ECMFA-TW), Paris, France 2010*, pp. 23–30.
[18] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event based traceability for managing evolutionary change," *IEEE Transactions on Software Engineering*, pp. 796–810, 2003.
[19] G. Spanoudakis, A. Zisman, E. Pérez-Miñana, and P. Krause, "Rule-based generation of requirements traceability relations," *Journal of Systems and Software*, vol. 72, no. 2, pp. 105–127, 2004.
[20] P. Mäder and O. Gotel, "Towards automated traceability maintenance," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2205–2227, 2012.
[21] L. G. P. Murta, A. van der Hoek, and C. M. L. Werner, "Continuous and automated evolution of architecture-to-implementation traceability links," *Automated Software Engineering Journal*, vol. 15, no. 1, pp. 75–107, 2008.
[22] B. Sharif and J. I. Maletice, "Using fine-grained differencing to evolve traceability links," in *International Symposium on Grand Challenges in Traceability (GCT07) 2007*, pp. 76–81.
[23] W. Uronkarn and T. Senivongse, "Change pattern-driven traceability of business processes," in *Proceedings of the International MultiConference of Engineers and Computer Scientists 2014 (IMECS 2014), Hong Kong*.
[24] F. Jouault, F. Allilaire, J. Bzivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, 2008.
[25] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *The International IEEE Enterprise Distributed Object Computing Conference (EDOC), IEEE Computer Society*, 2008, pp. 222–231.
[26] C. Brun and A. Pierantonio, "Model differences in the eclipse modeling framework," in *The European Journal for the Informatics Professional*, 2008, pp. 29–34.

Mohamed Yassine Haouam is an assistant professor in the department of Mathematics and Computer Science at the University of Tebessa, Algeria. He is a Ph.D. student at the University of Badji Mokhtar-Annaba (UBMA) and a member of the Laboratory of Mathematics, Informatics and Systems (LAMIS). He received his Magister degree in Computer Science from the University of Tebessa, Algeria in 2004. His current research interests include Model Driven Engineering, Traceability and Separation of Concerns.

Djamel Meslati is a professor in the department of Computer Science at the University of Badji Mokhtar-Annaba (UBMA), Algeria. he is the head of the Laboratory of Complex Systems Engineering (LISCO). His research interests include Software Development and Evolution Methodologies and Separation of Concerns.