Rule-based Impact Analysis for Heterogeneous Software Artifacts

Steffen Lehnert*, Qurat-ul-ann Farooq*, Matthias Riebisch†
*Department of Software Systems / Process Informatics, Ilmenau University of Technology
98684 Ilmenau, Germany
{steffen.lehnert, qurat-ul-ann.farooq}@tu-ilmenau.de
†Department of Informatics, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
riebisch@informatik.uni-hamburg.de

***Abstract*—Typical software engineering activities, such as program maintenance or reengineering, result in frequent changes of software which are often accompanied by unintended side effects. Consequently, research on impact analysis put forth approaches to assess the adverse effects of changes. However, understanding and implementing these changes is often aggravated by inconsistencies and dependencies between different types of software artifacts. Likewise, most impact analysis approaches are not able to detect the possible side effects of changes when different types of software artifacts are involved. We present an approach that combines impact analysis and multi-perspective modeling for analyzing the change propagation between heterogeneous software artifacts. Our approach assists developers with understanding the consequences of changes by identifying impacted artifacts and determining how they are affected. We utilize a model repository for combining UML models, Java source code, and JUnit tests by mapping them on a unifying meta-model. We introduce a novel impact propagation approach that analyzes dependencies between software artifacts according to the type of change which is applied upon them. Our approach is implemented by a set of impact propagation rules which are evaluated by a case study.**

***Keywords*-software evolution; impact analysis; multi-perspective modeling; horizontal traceability;**

I. Introduction

The lifecycle of software is characterized by frequent changes, such as bug fixes, the adaptation to new technology or the addition of new features. However, these changes are often accompanied by unintended side effects [1], such as new bugs or a loss of structure, which are also referred to as "ripple effects" [2]. Consequently, performing *software change impact analysis* [1] is required to assess the adverse effects of changes. Impact analysis enables developers to analyze the effects of changes prior to their implementation. It allows for a better planning of changes and eases decision making between alternative solutions based on their impacts.

Changes during software development apply to different types of artifacts, such as source code files, UML diagrams, and test cases. These artifacts provide different views on a system and tailor information according to different stakeholders [3]. As a result, the different artifacts are interdependent on each other [4], and changing one artifact requires changing related artifacts to avert further inconsistencies [5].

A recent study [6], however, indicated that only very few impact analysis approaches are able to determine the potential consequences of changing different types of artifacts. The majority of impact analysis approaches is still only focused on source code and source code files [6].

Research on horizontal traceability [7], multi-perspective modeling [8], and consistency checking [3], [9] addressed the problems and challenges arising from the interplay of different types of software artifacts. However, they do not yet provide support for impact analysis and the planning of future changes, nor do they facilitate the estimation of cost and time schedules for maintenance measures. Hence, multi-perspective impact analysis is required to support the evolution of software comprised of heterogeneous artifacts.

We combine impact analysis, multi-perspective modeling, and horizontal traceability analysis into one holistic approach supporting design models, source code, and test cases. Our approach combines UML models, Java source code, and JUnit tests through a unifying meta-model supplied by the Eclipse Modeling Framework [10] and a centralized model repository. We apply a traceability detection approach to uncover horizontal and vertical dependencies between the artifacts which are recorded as traceability links and are used for impact propagation.

Our main contribution is a novel rule-based approach for impact analysis, which analyzes the interplay of change operations and dependency relations between software artifacts to determine further change propagation. Finally, we present and discuss the design and results of an initial case study to evaluate our impact propagation approach.

Our paper is organized as follows. Section II discusses related work. Section III presents our approach and elaborates on our concepts of linking different models, dependency analysis, and impact propagation. The design and the results of an initial case study are presented in Section IV, before outlining future work and concluding the paper.

II. Related Work

In this section, we review impact analysis approaches for their support of heterogeneous artifacts. We then discuss how

research in traceability management, multi-perspective modeling, and multi-perspective consistency checking addresses the challenges of heterogeneous software artifacts.

Typical impact analysis approaches are focused on only one type of software artifact, such as source code or certain UML diagrams. A previous study [6] revealed, that from 150 studied impact analysis approaches only 19 are able to analyze at least two types of artifacts. Various techniques have been proposed to assess the propagation of impacts, such as program slicing [11], call graph analysis [12], analysis of execution traces [13], [14], static execute after relations [15], impact rules [16]–[18], information retrieval [19]–[21] or the mining of software repositories [22]–[24]. However, most techniques are limited to only one type of artifact. For example, a call graph analysis cannot be applied on requirements' specifications. Due to this limitation, most of the proposed approaches are not able to detect impacts in heterogeneous software artifacts. This, however, is typical for most software as discussed in the previous section.

Ibrahim *et al.* [25] proposed an approach for impact analysis spanning classes, packages, tests, and requirements. The authors establish traceability relations between software artifacts which are based on similar names, domain knowledge, and explicit relationships. However, they provide semi-automated traceability analysis only, which limits the applicability of their approach. They further neglect widely used UML diagrams and fine-grained code artifacts.

Most impact analysis approaches are further limited by the amount of change types they support. They treat different types of changes equally and assume that they result in the same consequences. Only few works, such as the approach of Keller *et al.* [17], [18], treat different types of changes separately during the impact analysis process. However, their set of change operations is not comprehensive either, as only a subset of operations discussed in [26] is covered.

The problem of multi-perspective impact analysis is closely related to challenges that researchers are confronted with in the fields of multi-perspective modeling, vertical and horizontal traceability analysis, and multi-perspective consistency checking, namely: highly interdependent, but heterogeneous artifacts with different structure and purpose.

Multi-perspective consistency checking [9] encompasses approaches developed for maintaining different types of software artifacts in a consistent state. Fradet *et al.* [9] presented a framework for analyzing architectures comprised of multiple views, which are linked by traceability relations. Sunetnanta and Finkelstein [3] explored the reasons why software development requires different perspectives, models, and diagrams from an end-users point of view. They propose a multi-perspective viewpoint framework, which is used for linking different models. Likewise, Olsson and Grundy [4] utilized traceability relations between different artifacts to adapt them to changes, thereby maintaining the overall consistency of the system. Eramo *et al.* [8] presented a framework for multi-view modeling utilizing multiple independent views and viewpoints which are connected by correspondence relations.

Research conducted by Lindvall and Sandahl [7], [27] outlined the applicability of vertical and horizontal traceability relations for impact analysis, since existing traceability relations can be used for ripple effect analysis [28]. Traceability links allow for connecting different types of software artifacts [29], which makes them suitable for expressing dependencies between heterogeneous software artifacts.

De Lucia *et al.* [5] discussed the need of traceability support for impact analysis and the two main problems of traceability analysis: link recovery and link evolution. In our approach, we only deal with link recovery. We refer to the work of Mäder [30] for a detailed discussion of the maintenance problem and to our previous work [31]–[33] for further discussions of traceability recovery approaches.

Rochimah *et al.* [34] reviewed traceability detection approaches based on various criteria, where the applicability for software evolution, the support for different types of software artifacts, and their degree of automation are of special interest for our approach.

Imtiaz *et al.* [35] analyzed various traceability techniques for their support of multi-perspective impact analysis, which is based on a similar classification of techniques as applied by Rochimah *et al.* They define a set of criteria for comparing approaches, where the support for horizontal and vertical traceability is of particular importance for our work.

We can draw the following conclusions. First, a unifying representation or framework on which all the different artifacts can be mapped upon eases multi-level analysis [8], [9]. Secondly, it requires adequate means for connecting the heterogeneous artifacts, i.e. horizontal and vertical traceability links [7], [27], to enable further analysis. According to results of Rochimah *et al.* [34] and Imtiaz *et al.* [35], a rule-based traceability mining technique is suitable for dependency detection among the different types of artifacts, which can also be fully automated. Last, our impact analysis approach should be able to address different types of changes adequately [18] and support the operations defined in [26].

III. Multi-perspective Impact Analysis

A. Problem Space and Requirements

We consider the perspectives of requirements, architecture, source code, and test cases to provide a holistic approach providing support for different development stages. Hence, we support UML models, Java source code, and JUnit tests. Figure 1 illustrates these perspectives, their respective software artifacts, and potential dependencies between them (arrows). Analyzing the impact propagation between these heterogeneous software artifacts holds two major challenges: (1) interconnecting the different types of software artifacts and (2) propagating changes across them. Based on our problem space, the challenges discussed above,

and conclusions drawn from related work, we define our requirements as follows:

1) Enable impact analysis of dynamic and static UML models, requirements, source code, and test cases.
2) Combine the artifacts on a common base to use the same analysis approach for all types of artifacts.
3) Detect and record dependencies between software artifacts for impact analysis.
4) Enable developers to easily understand and retrace the propagation of changes.

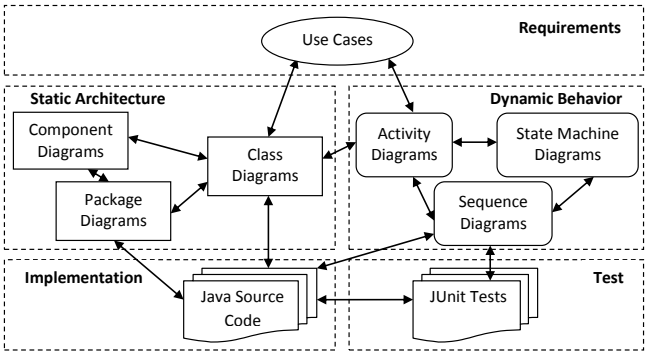

Figure 1. Views and artifacts considered by our approach

The following section presents an overview of our approach and answers how we fulfill our four requirements.

B. Overview

Our first two requirements demand for a homogenous treatment of the different types of artifacts. This, however, requires a unifying framework supplying a common meta-model on which all types of artifacts can be mapped upon. Furthermore, it requires a centralized model repository for storing the artifacts. Thus, we build our approach upon the *Eclipse Modeling Framework* (EMF) [10] and the EMF-based model repository *EMFStore*[1]. EMF implements a unifying meta-model and allows us to map all artifacts to EMF-based models. Section III-I further explains the details of this mapping, the model repository, and the model import.

After the artifacts are unified, we apply a rule-based approach for traceability discovery to record dependency relations between them as traceability links. This approach extends our previous work on dependency detection [31]–[33], which is based on an analysis of related work on traceability discovery, such as information retrieval approaches. Section III-C discusses the structure of our dependency relations, while Section III-D elaborates on the structure, origin, and evaluation of our dependency detection rules. Both sections therefore contribute towards fulfilling our third requirement.

The diversity of software artifacts requiring impact analysis directly influences our choice of a suitable impact analysis algorithm. Most code-based techniques, such as slicing or call graph analysis, are not applicable for the different types of artifacts. Dependency analysis, on the other hand, can be applied on any kind of dependency relation which exists between software artifacts. However, pure dependency analysis is too imprecise and produces too many false-positives [36]. Therefore, we propose an improved version of dependency analysis resulting in less false-positives as confirmed by our case study (see Section IV), which is based on a novel propagation approach.

Our impact propagation technique is based on the type of dependency which exists between two EMF-models and the type of change which is applied on one of them. The underlying hypothesis is that the interplay of change type, dependency type, and the types of involved artifacts determines if and how a change ripples to related artifacts. This interplay is implemented by a set of impact propagation rules, which are applied in a recursive manner until all impacted elements have been identified. Section III-F illustrates the structure and origin of our propagation rules, whereas Section III-G further explains our propagation concept. Our propagation rules are able to analyze any kind of EMF-model and thus contribute towards our first requirement.

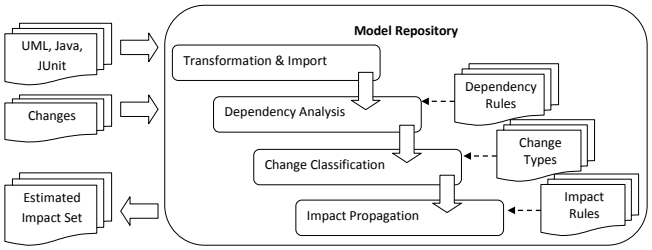

Figure 2. Overview of our approach.

Our concept of change types is built on previous research on the role of change types in software evolution [26]. We distinguish between atomic and composite changes, which are used for modeling real change activities. Section III-E summarizes this concept and provides an example.

In summary, our entire approach consists of four major steps, which are also illustrated by Figure 2:

1) Transformation of artifacts into unified EMF-based models and import into model repository.
2) Rule-based dependency analysis.
3) Classification of change type.
4) Rule-based computation of impact propagation.

The tool implementing our approach is introduced in Section III-I. We also discuss the extensibility of our approach regarding the integration of new modeling and programming languages in Section III-I.

C. Dependency Relations

We record dependencies between software artifacts as traceability links to fulfill our third requirement. A traceability link consists of three attributes: the source model, the

[1] http://eclipse.org/emfstore/

target model, and the type of relation. Our approach requires explicit types of dependency relations, which are based on a review of related work and are clustered according to their purpose:

- *Structural relations* expressing the structure of artifacts, e.g. *Contains* or *PartOf*.
- *Behavioral relations* expressing the dynamic relations between artifacts, e.g. *Calls* or *Tests*.
- *Conditional relations* expressing constraints and conditions between artifacts, e.g. *Satisfies* or *Requires*.
- *Lifecycle relations* expressing evolutionary development, e.g. *EvolvesTo* or *Replaces*.
- *Similarity relations* expressing similarities and equivalences, e.g. *Overlaps* or *SimilarTo*.

Our dependency relation types are based on dependencies within the object oriented design paradigm (e.g. inheritance relationships), related work on requirements traceability [37], [38], traceability recovery [39]–[41], impact analysis [17], [18], and consistency management [4], [42]. Further examples of relation types are displayed by Figure 3, which presents an excerpt of our case study and is discussed in Section III-H. A comprehensive list of all dependency types can be obtained from our project website [43].

D. Dependency Detection Rules

The following section explains our dependency detection rules which record dependencies as traceability links.

1) Origin of Rules: We are currently using a set of 130 rules to detect dependency relations between different UML models, OWL models, URN models, Java source code, and JUnit tests which are based on four different sources:

- *Relations defined in the meta-models of artifacts.* The object-oriented paradigm for example contains a variety of dependency relations, e.g. *inheritance*-relations between classes or *implementation*-relations between classes and interfaces. Rules can be established which transform these dependencies into explicit links.
- *Correspondences according to design methodologies.* For object-oriented development for instance, there is a general rule of equivalence between design and code. For example, equivalences exist between UML and Java classes or between UML and Java packages.
- *Related work on rule-based traceability detection.* For example, Jirapanthong and Zisman [41] provide rules for linking different UML models, whereas Filho *et al.* [44] provide rules for linking UML and URN models which are reused in our approach.
- *Similarities and overlappings in names and identifiers of artifacts.* Although less precise than above discussed relations, similar names can also provide useful traceability relations, e.g. when a use case actor and a class or a use case and a method share the same name.

All rules can be obtained from our project website [43] and are ready to be used within our prototype tool, which is introduced in Section III-I.

2) Structure of Rules: Our dependency detection rules resemble typical SQL-queries and consist of three parts:

- Element-Definition: which types of artifacts are addressed by the rule?
- Query-Definition: how are the artifacts related?
- Result-Definition: what should be done with them?

Within the query-part, our rules analyze the structure, relations, and names of model elements to determine traceability links. For further details, we refer to our previous work [32], [33]. An example demonstrating the structure of our rules and possible query-operators is illustrated by Listing 1.

```
<rule id="TraceRule012">
  <elements>
    <element alias="e1" type="Method"/>
    <element alias="e2" type="Method"/>
    <element alias="e3" type="Class"/>
    <element alias="e4" type="Interface"/>
  </elements>
  <query>
    <condition value="RELATED_TO(e3,'Implements',e4)"/>
    <condition value="PARENT_OF(e3,e1)"/>
    <condition value="PARENT_OF(e4,e2)"/>
    <condition value="e1::name=e2::name"/>
  </query>
  <results>
    <create_link from="e1" to="e2" type="Implements"/>
  </results>
</rule>
```

Listing 1. A rule linking corresponding methods of interfaces and classes.

This rule establishes a link between the declaration and implementation of a method. It demonstrates how we exploit the structure ("PARENT_OF"-conditions), names ("="-condition), and relations ("RELATED_TO"-condition) which exist between models when searching for dependency relations. Our rules offer a variety of query-statements, which can be nested by AND, OR, NOT, and XOR.

3) Evaluation: We performed an evaluation of our dependency detection rules using two case studies [31]. We applied our rules on a robot control software and on our own prototype tool EMFTrace [32]. The robot control software was developed by more than 20 researchers and programmers, while five persons were involved in the development of EMFTrace. All researchers and developers manually analyzed the cases to provide an oracle for computing precision and recall. Finally, our dependency detection approach achieved a mean precision of 0.84 and a mean recall of 0.85 [31], which is comparable to similar approaches [41].

E. Supported Change Types

As we consider the type of change essential for determining its impact, we investigated the usage of change types in software evolution, maintenance, and regression testing [26]. Using a graph-based representation of software artifacts, we derived a taxonomy of change types comprised

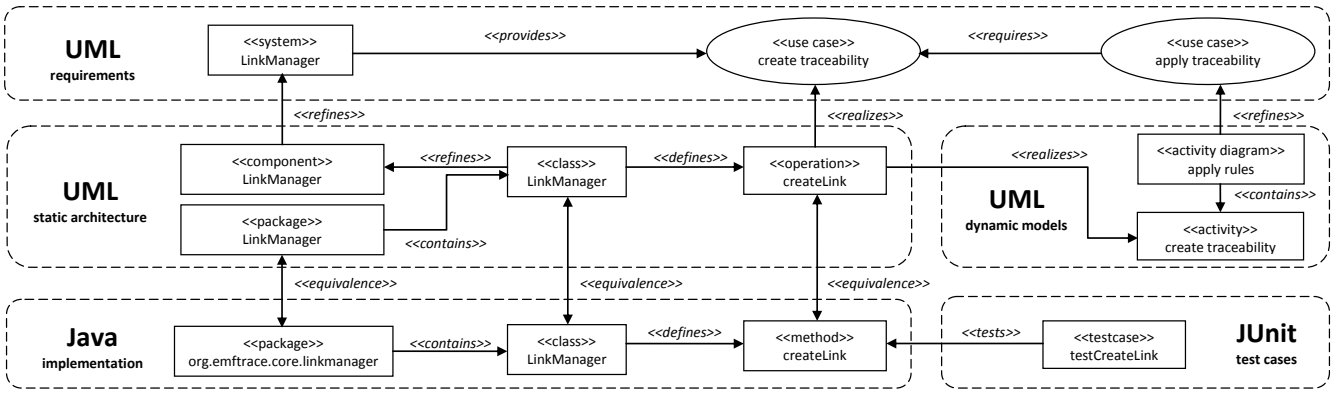

Figure 3. An excerpt of our case study demonstrating our dependency concept.

of atomic operations ($add, delete, update$) and composite operations ($move, replace, split, merge, swap$), where the latter may consist of sequences of atomic and composite operations. For example, the composite operation "splitting a class" can be modeled by adding a new class and moving all required attributes and methods to this new class. Each *move*-operation itself is again modeled by a *delete* and *add*-operation. Based on our taxonomy, we defined a set of 140 change types, which range from "simple changes" like *deleting a method from a class* to complex operations, such as *merging UML components*. A complete list of change types can be obtained from our website [43].

F. Impact Propagation Rules

In this section, we present our impact propagation rules, before explaining our propagation concept in Section III-G.

1) Origin of Rules: Our propagation rules are derived from our previously defined dependency detection rules. Depending on the types of artifacts addressed by a rule, we extend the rule to react on one of the change types presented in Section III-E. We illustrate this process using *TraceRule012* as an example (see Listing 1), which links method-declarations with their respective method-implementations. Consequently, *TraceRule012* should be extended to react on changes that affect both elements. Thus, we derived rules to react on name changes, return type changes, the addition or deletion of parameters, modifier changes, the deletion of the method, and moving the method to another class or interface. For example, *ImpactRule041* (see Listing 2) implements the "rename method"-case. The conditions defined in the *query*-part check if the applied change type matches "rename method", if the change was applied on the method-declaration, and if the method-declaration is "Implemented" by a method of a class. If all conditions are satisfied, the name change is propagated to the method-implementation.

However, we do not consider all possible types of changes for all types of artifacts. Some combinations are not relevant in practice, e.g. *splitting* or *merging* attributes of classes. Finally, we derived a set of 180 impact rules, which can also be obtained from our website [43].

2) Structure of Rules: The structure of our propagation rules is equivalent to our dependency detection rules. They offer the same query-operators and utilize the same query-mechanisms. The only difference is that they create impact reports instead of traceability relations. Thus, they are able to query any type of artifact converted into an EMF-based model, and thereby fulfill our first requirement.

```
<rule id="ImpactRule041">
  <elements>
    <element alias="e1" type="Method"/>
    <element alias="e2" type="Method"/>
    <element alias="e3" type="AtomicChangeType"/>
  </elements>
  <query>
    <condition value="e3::name='Rename method'"/>
    <condition value="e3::target=e2"/>
    <condition value="RELATED_TO(e1,'Implements',e2)"/>
  </query>
  <results>
    <report_impact from="e2" to="e1" type="e3::name"/>
  </results>
</rule>
```

Listing 2. A rule to react on the change of a method's name.

Our rules allow developers to easily retrace the propagation of changes by inspecting generated impact reports and the relations between models, which fulfills our fourth requirement.

G. Impact Propagation Approach

We determine the propagation of impacts by analyzing the type of dependency relation which exists between two model elements and the type of change which is applied on one of them. This approach is implemented by a set of propagation rules, which are applied in a recursive manner and were introduced in the previous section.

Each rule receives the following input: the changed element, the type of change, and a list of all related elements. This input is then processed according to the query-conditions defined in the rule to remove all related elements that do not adhere to these conditions. As a result of the

query-processing, a list of all impacted element(s) along with the resulting change type(s) is computed. This output is then again fed into the impact analysis process. The recursive propagation continues as long as there is output matching the required input of another rule. It is important to note, that the change type resulting from a rule must not necessarily equal the change type which is fed into the rule. For example, merging two classes may result in all attributes that store instances of one of the former classes having to change their data types or being deleted otherwise.

Cyclic dependencies between software artifacts may lead to infinite loops during impact propagation. We address this problem by maintaining two lists storing 3-tuples (changed element, change type, impacted element) to record the current progress of impact propagation, which is inspired by the *A**-pathfinding algorithm [45]. The *ClosedList* contains all already "explored" 3-tuples or impact paths. The *OpenList* contains all 3-tuples which might lead to new impact paths and should be further explored. If a possible new impact path is found, the *ClosedList* is searched for a tuple containing the same elements and change types. If such a tuple is found, further propagation on this path is stopped. Algorithm 1 illustrates our propagation approach in pseudocode.

Algorithm 1 Recursive change propagation

```
1: openList.add(new Tuple(changedModel, change, relatedModels));
2: while (!openList.isEmpty()) do
3:     List < Tuple > tmp = executeRules(openList.get(0));
4:     for (i = 0; i < tmp.size()) do
5:         if (!containsTuple(openList, tmp.get(i))) then
6:             openList.add( tmp.get(i) );
7:         end if
8:     end for
9:     if (!containsTuple(closedList, openList.get(0))) then
10:        closedList.add( openList.get(0) );
11:        openList.remove(0);
12:    end if
13: end while
```

In contrast to existing dependency-based approaches, our propagation strategy differs regarding certain key aspects.

First, we do not propagate changes across any dependency relation regardless of their type, which limits the size of impact sets to be computed.

Secondly, our propagation approach does not involve any cutoff-distance to limit the change propagation, such as in the work of Bohner [36] or Briand *et al.* [16]. In our approach, only the interplay of change type and dependency type determines the propagation of changes. This hypothesis is underpinned by results of an initial case study presented in Section IV. Furthermore, defining a reasonable cutoff-distance becomes increasingly difficult when multiple perspectives are involved. A cutoff-distance set too close results in many impacted elements being missed. On the other hand, a cutoff-distance set too far away will result in too many artifacts being considered as impacted.

Thirdly, our rules determine *how* impacted elements are actually impacted; respectively how they should be changed. This offers two benefits for developers: They are aware of how to change the software, and they are able to discuss alternative solutions based on the size of their impact sets *and* the types of changes required by a certain solution.

H. Illustrative Example

For illustrative purpose, we discuss an example of recursive type-based impact propagation. The example is taken from our case study (see Section IV) and is depicted by Figure 3. In this example, the UML component *LinkManager* is renamed and all impacted elements shall be identified. In a first step, the propagation rules analyze all directly related elements of the component. In this case, there are two related and thus potentially impacted elements: a class and a system with the same name. As both relations are of the type "Refines", both carry the name change. There is no further direct dependency and we have to inspect previously identified impacts recursively. When examining the system *LinkManager*, the remaining "Provides"-relation will not propagate the rename-operation and there is no further recursive propagation. The UML class *LinkManager*, however, is connected to a Java class *LinkManager* ("Equivalence"-relation), and with an UML package *LinkManager* ("Contains"-relation). The "Contains"-relation does not propagate the rename-operation and the inspection of the package is complete. The Java class, however, is impacted by the name change and should as well be renamed. The impacted Java class is further related to a Java package ("Contains"-relation) and to a Java method ("Defines"-relation), which both do not propagate the name change any further.

I. Tool Support

The presented approach is implemented by our prototype tool EMFTrace [43]. The tool was initially developed for dependency detection between different types of software artifacts [31]–[33], and is available as an open source project. EMFTrace is based on EMF and the EMFStore model repository (see Figure 4), which supply a homogenous meta-model and a model repository respectively.

Our tool supports Java source code, JUnit test cases, UML models, URN models, OWL models, and feature models. Additional modeling or programming languages can be added to EMFTrace by providing EMF-based meta-models, which is explained by tutorials hosted on our project website [43]. The required Ecore-mapping and model import is fully automated and provided through the user interface of EMFTrace. XSLT templates are used for converting instances of UML, OWL, URN, and feature models, while Java source code files and JUnit test cases are converted using MoDisco[2].

[2]http://www.eclipse.org/MoDisco/

EMFTrace further extends EMFStore by adding features for rule-based dependency detection, dependency visualization, distance-based impact analysis, and rule-based impact analysis. All features are fully automated; however, manual effort is required for creating and maintaining dependency detection and impact analysis rules. Support for impact analysis is currently provided for Java, JUnit, and UML only. However, it can be extended to encompass other modeling languages by providing suitable impact propagation rules.

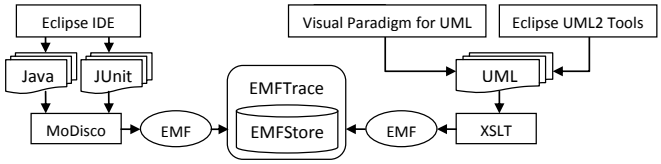

Figure 4. Our prototype tool EMFTrace.

IV. Case Study and Evaluation

The following presents an initial case study which was conducted to test the applicability of our approach and to evaluate its performance. We followed the guidelines of Runeson and Höst [46] for designing the study, and the approach of Basili *et al.* [47] for defining and quantifying our metrics. We discuss our research questions, the design of the study, achieved results, and possible threats to validity. All the data used throughout our case study can be obtained from the download section of our project website [43].

A. Objective and Research Questions

The purpose of our initial study is to demonstrate the applicability of our approach and its benefits for developers when applied on heterogeneous software artifacts. Our goals are as follows. First, we seek confirmation for our hypothesis that the impact of a change depends on the type of change, the type of dependency relation between involved artifacts, and the type of involved artifacts. Secondly, we want to apply our approach on a system of realistic size and complexity. Last, we are interested in the performance of our approach regarding recall and precision. In conclusion, we derived the following two research questions.

RQ1: *Does the interplay of change type, dependency type, and artifact type determine further change propagation?* We analyze, if our propagation strategy is able to determine the impact sets defined by our oracle. We further investigate, if our propagation strategy results in fewer missed impacts and false-positives than existing dependency-based propagation approaches. We accomplish this by computing and comparing impact sets using (1) our approach and (2) distance-based propagation using a cutoff distance. According to results of Hassaine *et al.* [48], we use a cutoff distance of 2 for the second case. Based on the computed impact sets, we determine the precision and recall for both techniques.

RQ2: *Is our change propagation approach able to predict impacts between heterogeneous artifacts?* We apply our approach on a system comprised of different, highly interdependent software artifacts, such as Java source code, JUnit test cases, and several UML diagrams. Different artifacts are changed according to our taxonomy of change types [26] and the propagation of those changes across the different perspectives is analyzed. The amount of false-positives and missed impacts is analyzed for each type of software artifact, to assess the applicability of our approach on different types of software artifacts.

B. Design and Subject of the Study

According to the goals of our study, we require a case for which UML models, Java source code, and JUnit test cases are available. To answer both research questions, we apply a series of changes on the chosen case. We manually analyze the case for the impacts of each change to provide an oracle for comparing the computed impact sets. Thus, we require decent knowledge of the studied system, its structure, and dependencies to be able to carry out the changes. We analyzed nine open source systems, including a render engine[3], a steering library for autonomous AI agents[4], and our own prototype tool [43]. Our main requirement, the availability of different types of software artifacts, turned out to be the limiting factor when deciding for a case, as most free accessible projects only provide source code. Finally, we decided to apply our approach on our own prototype tool EMFTrace for three reasons. First, we have sufficient knowledge to evaluate the impacts of changes, to supply the oracle for comparing our results. Secondly, various UML diagrams (component, class, package, use case, and activity diagrams), Java source code, and JUnit tests are available for analysis, altogether 16,500 elements. Last, we have full access to all available data and the rights to publish them.

1) Designing the Change Operations: The change operations were designed and chosen as follows. For each type of artifact provided by our case, we derived a list of possible change operations to obtain all relevant combinations of artifact types and change types. For example, *class*-changes encompass: *add method*, *delete method*, *add attribute*, *delete attribute*, *rename class*, *merge with other class*, *split class* (i.e. extract sub-class), *move to other package*, *change superclass*, and *change implemented interface*. This procedure was repeated for each type of artifact provided by the case and resulted in 48 changes. The ratio of changes per type of software artifact is further summarized by Table I.

2) Building the Oracle: Obtaining the oracle for each change operation was achieved by manually applying the change on the system and inspecting its impact. For each of the 48 change operations, we analyzed the source code

[3]http://www.ogre3d.org/

[4]http://opensteer.sourceforge.net/

Artifacts and Changes				Results of Rule-based Propagation				Results of Distance-based Propagation			
Type	# of Changes	Change IDs	$\|AIS\varnothing\|$	$\|EIS\varnothing\|$	Precision	Recall	F_1-score	$\|EIS\varnothing\|$	Precision	Recall	F_1-score
Component	9	C01 - C09	25	25	0.9783	0.9231	0.9498	58	0.1640	0.4506	0.2404
Package	7	C10 - C16	10	8	0.9365	0.9080	0.9220	5	0.4285	0.5751	0.4910
Class/Interface	10	C17 - C26	19	21	0.9556	0.8809	0.9167	208	0.0649	0.7027	0.1188
Method	8	C27 - C34	4	4	1.0000	0.9000	0.9473	138	0.0386	0.8463	0.0738
Methodparameter	4	C35 - C38	3	3	1.0000	1.0000	1.0000	50	0.0196	0.3333	0.2468
Attribute	6	C39 - C44	7	6	1.0000	0.9023	0.9486	26	0.0958	0.4220	0.1561
Use case	4	C45 - C48	11	11	1.0000	0.9210	0.9588	48	0.1339	0.6840	0.2239

Table I. CHANGES AND RESULTS OF OUR CASE STUDY.

and UML diagrams for change propagation. Each artifact identified by manual inspection was then added to the oracle impact set for comparing the results of our approach.

C. Results and Discussion

The following discusses the results of our study regarding our two research questions. Due to space constraints, we are not able to discuss the impact sets of all 48 change operations in detail. However, a list of all changes along with the computed impact sets can be obtained from the download section of our project website [43]. The source code, test cases, and all UML diagrams used within our study can also be obtained from the same website.

1) ***RQ1 -*** *Does the interplay of change type and dependency type determine further change propagation:* To answer this question, we compare the impact sets computed by our approach against our oracle and with the impact sets computed by distance-based propagation.

First, we need to compare the sizes of our impact sets with those of the oracle. There is a maximum difference of 20% between the sizes of our impact sets and the oracle (*Package*-changes, see Table I), whereas impact sets computed by distance-based propagation contain up to 2100% more elements (*Class*-changes, see Table I). In average, the sizes of our impact sets correspond to those of the oracle, while those of distance-based propagation vary considerably.

Secondly, we have to analyze our impact sets for false-positives to determine the precision of our approach. Only four out of 48 computed impact sets contain false-positives at all (*C07*, *C08*, *C15*, and *C24*; see Figure 5). In contrast, 44 out of 48 impact sets computed by distance-based propagation consist of more than 40% false-positives (see Figure 5). While our type-based propagation approach achieved an average precision of more than 80%, distance-based propagation resulted in a mean precision of less than 20%. Results of type-based propagation are also much more "stable" than those computed by distance-based propagation. There are only two outliers for precision using type-based propagation (*C15*: 0.556, *C24*: 0.614; see Figure 5), which, however, are still well above the average precision computed by distance-based propagation.

Thirdly, we need to determine the recall of our approach by analyzing the impact sets for missed impacts. Compared to the average recall of type-based propagation, there is only one outlier (*C21*: 0.294; see Figure 5) which is caused by yet not recorded dependency relations between attributes and their getter and setter methods. In contrast, impact sets of distance-based propagation show a greater overall variance and missed more impacted elements (see Figure 5). The recall of distance-based propagation could have been improved by applying a larger cutoff distance, e.g. 3 or 5. This, however, would further reduce the precision by introducing too many false-positives.

In conclusion, our approach was able to determine the impact sets defined by our oracle with an average precision and recall of well above 80%. Our approach identified only few false-positives and did not overestimate the propagation of changes. From a practical point of view, this means that our approach can provide developers with a solid estimation of the impact of a change. Hence, we are able to positively answer **RQ1**.

2) ***RQ2 -*** *Is our change propagation approach able to predict impacts between heterogeneous artifacts:* We applied a series of changes on different types of software artifacts, compared the computed impacts sets against our oracle, and finally measured the achieved precision and recall. Table I summarizes the distribution of changes across the different types of artifacts. The changed artifacts represent typical artifacts from different perspectives and development phases, such as use cases (requirements), component diagrams (static architecture) or class diagrams (fine-grained architecture, implementation, tests). As shown by Table I and discussed for **RQ1**, our rule-based propagation approach was able to determine the impacts of changes with reliable and stable results for any type of software artifact. Although many changes had affects on different types of artifacts simultaneously, our approach was able to correctly determine the change propagation across the different perspectives. Therefore, we are also able to positively answer **RQ2**.

In conclusion, our approach was able to determine impact sets which correspond to those of the oracle. Our propagation strategy provided reliable and stable figures for precision and recall. Furthermore, our study indicates that our approach is applicable on heterogeneous artifacts, and produces significantly less false-positives than distance-based propagation. Our approach also required less than five minutes for computing the impacts sets, while manual analysis took more than 16 hours. Consequently, it reduces

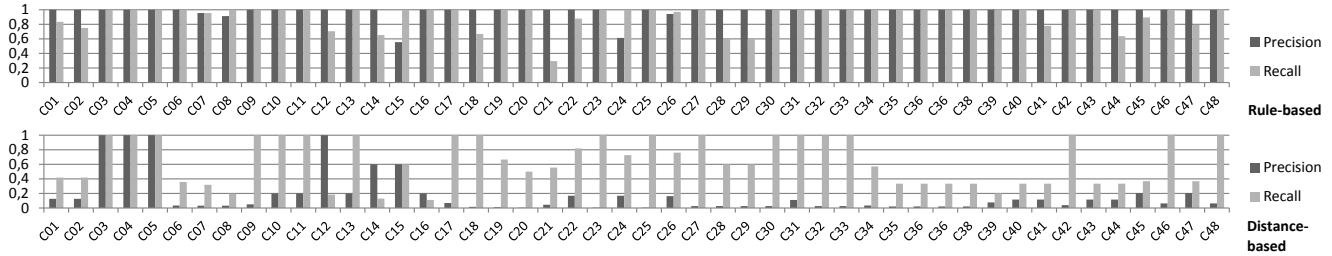

Figure 5. Precision and recall of impact sets computed by rule-based propagation (upper diagram) and distance-based propagation (lower diagram).

the effort for developers when examining and understanding the affects of changes, thus assisting with changing software.

D. Threats to Validity

There are four categories of potential threats limiting the validity of our initial case study [18], [46].

Construct Validity: do we measure what was intended? We measure the impact of a change according to the amount of impacted artifacts and use formulas for precision and recall as provided in related work.

Internal Validity: are there unknown factors which might affect the causal dependencies? We were able to correctly classify each change applied on our case study according to our taxonomy of change types [26]. Thus, we can be sure that the correct set of rules was chosen by our tool.

External Validity: to what extent it is possible to generalize our findings? We analyzed a variety of UML diagrams, Java source code, and JUnit test cases to test our rules with a wide range of possible data sources. However, the studied case provided almost complete and very detailed architectural diagrams, whereas other applications might lack such details. Due to time constraints, we were further not able to test all combinations of change types and artifact types. However, our initial study covered a variety of change operations, such as merging packages or the deletion of use cases.

Reliability: are the results dependent on the researcher and the tools? Although we supplied the oracle to compare the output of our approach, we can limit our influence on the results due to our experience in developing, maintaining, and analyzing the studied case. Our dependency detection technique may also influence the results of our case study. However, we evaluated the performance of our dependency detection technique in previous research using two case studies and achieved reliable results for both [31].

V. Conclusion and Future Work

Software development requires different types of artifacts to reflect different views on the system and to express different concerns. Frequent changes introduce inconsistencies to these artifacts, if changes are not addressed adequately among them. This results in further defects, decreased maintainability, and increased gaps between high-level design and implementation. However, most impact analysis approaches are not applicable for heterogeneous artifacts, while existing techniques for multi-level modeling and multi-perspective consistency checking are not able to predict future changes and their impacts.

We presented an approach combining impact analysis with the concept of multi-level modeling, to assist with maintaining software through impact analysis of different UML models, Java source code, and related JUnit tests. Our impact analysis approach relies on change propagation rules, which analyze dependency relations between software artifacts according to the type of change which is applied upon them. We illustrated how a dependency detection technique is used to elicitate vertical and horizontal dependency relations between the various artifacts, and how our approach is implemented by our prototype tool EMFTrace.

Results of an initial case study confirmed that our approach is able to determine impacted artifacts with reliable precision and recall. Our approach also required significantly less time when compared to manual analysis. Furthermore, it determined the propagation of impacts more reliably than distance-based dependency analysis. Impact sets computed by our approach were more precise, missed fewer impacted artifacts, and provided a greater overall stability. Our study also revealed that rule-based impact propagation can be applied in the context of heterogeneous software artifacts, as different types of software artifacts were changed and the propagation of impacts was correctly determined.

Our approach offers several opportunities for further research. First, a more systematic investigation of dependency types is required, which is currently accomplished by a literature review. Secondly, we need to refine and add additional dependency detection rules to elicitate further, yet missing dependencies, as revealed by our case study. Thirdly, we are currently extending the dependency analysis of Java source code to the level of program statements, to allow for more fine-grained couplings with dynamic UML models.

References

[1] S. A. Bohner and R. S. Arnold, *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Publications Tutorial Series, 1996.

[2] S. S. Yau, J. S. Collofello, and T. M. McGregor, "Ripple effect analysis of software maintenance," in *Computer Software and Applications Conference (COMPSAC '78)*, 1978, pp. 60–65.

[3] T. Sunetnanta and A. Finkelstein, "Automated consistency checking for multiperspective software specifications," in *Workshop on Advanced Separation of Concerns*, 2001.

[4] T. Olsson and J. Grundy, "Supporting traceability and inconsistency management between software artifacts," in *Intl. Conf. on Software Eng. and Applications*, 2002, pp. 63–78.

[5] A. De Lucia, F. Fasano, and R. Oliveto, "Traceability management for impact analysis," in *Frontiers of Software Maintenance (FoSM 2008)*, 2008, pp. 21–30.

[6] S. Lehnert, "A review of software change impact analysis," Ilmenau University of Technology, Tech. Rep., 2011.

[7] M. Lindvall and K. Sandahl, "Practical implications of traceability," *Softw. Pract. Exper.*, vol. 26, pp. 1161–1180, 1996.

[8] R. Eramo, A. Pierantonio, J. R. Romero, and A. Vallecillo, "Change management in multi-viewpoint system using asp," in *5th Int. Workshop on ODP for Enterprise Computing (EDOC 2008)*, Munich, Germany, 2008, pp. 19–28.

[9] P. Fradet, D. Métayer, and M. Périn, "Consistency checking for multiple view software architectures," *Lecture Notes in Computer Science*, vol. 1687, pp. 410–428, 1999.

[10] "Eclipse Modeling Framework (EMF)." [Online]. Available: http://www.eclipse.org/modeling/emf/

[11] L. Vidács, A. Beszédes, and R. Ferenc, "Macro impact analysis using macro slicing," in *2nd Intl. Conference on Software and Data Technologies*, 2007, pp. 230–235.

[12] B. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Workshop on Program analysis for software tools and engineering (PASTE '01)*, 2001, pp. 46–53.

[13] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *9th European Conf. on Software Eng.*, 2003, pp. 128–137.

[14] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Intl. Conference on Software Engineering (ICSE '03)*, 2003, pp. 308–318.

[15] A. Beszédes, T. Gergely, J. Jász, G. Tóth, T. Gyimóthy, and V. Rajlich, "Computation of static execute after relation with applications to software maintenance," in *IEEE Intl. Conf. on Software Maintenance (ICSM 2007)*, 2007, pp. 295–304.

[16] L. Briand, Y. Labiche, L. O'Sullivan, and M. Sówka, "Automated impact analysis of UML models," *Journal of Systems and Software*, vol. 79, pp. 339–352, 2006.

[17] A. Keller, H. Schippers, and S. Demeyer, "Supporting inconsistency resolution through predictive change impact analysis," in *6th Intl. Workshop on Model-Driven Engineering, Verification and Validation*, 2009.

[18] A. Keller and S. Demeyer, *Change Impact Analysis for UML Model Maintenance*. IGI Global, 2011, ch. 2, pp. 32–56.

[19] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Identifying the starting impact set of a maintenance request: A case study," in *4th European Conference on Software Maintenance and Reengineering*, 2000, pp. 227–230.

[20] S. Vaucher, H. Sahraoui, and J. Vaucher, "Discovering new change patterns in object-oriented systems," in *15th Working Conf. on Reverse Engineering (WCRE '08)*, 2008, pp. 37–41.

[21] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[22] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, September 2004.

[23] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, June 2005.

[24] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "Using multivariate time series and association rules to detect logical change coupling: an empirical study," in *26th IEEE Intl. Conf. on Software Maintenance (ICSM 2010)*, 2010.

[25] S. Ibrahim, N. B. Idris, M. Munro, and A. Deraman, "Integrating software traceability for change impact analysis," *The Intl. Arab Journal of Information Technology*, vol. 2, no. 4, pp. 301–308, 2005.

[26] S. Lehnert, Q.-U.-A. Farooq, and M. Riebisch, "A taxonomy of change types and its application in software evolution," in *19th Annual IEEE Intl. Conference on the Engineering of Computer Based Systems*, 2012, pp. 98–107.

[27] M. Lindvall and K. Sandahl, "Traceability aspects of impact analysis in object-oriented systems," *Journal of Software Maintenance: Research and Practice*, vol. 10, pp. 37–57, 1998.

[28] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software and Systems Modeling*, vol. 9, no. 4, pp. 529–565, 2010.

[29] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni, "Model traceability," *IBM Systems Jounal*, vol. 45, no. 3, pp. 515–526, July 2006.

[30] P. Mäder, *Rule-Based Maintenance of Post-Requirements Traceability*. MV-Verlag, Münster, 2010.

[31] S. Lehnert, "Softwarearchitectural design and realization of a repository for comprehensive model traceability," Master's thesis, Ilmenau University of Technology, 2010.

[32] S. Bode, S. Lehnert, and M. Riebisch, "Comprehensive model integration for dependency identification with EMFTrace," in *1st Intl. Workshop on Model-Driven Software Migration (MDSM 2011) and the 5th Intl. Workshop on Software Quality and Maintainability (SQM 2011)*, 2011, pp. 17–20.

[33] M. Riebisch, S. Bode, Q.-U.-A. Farooq, and S. Lehnert, "Towards comprehensive modelling by inter-model links using an integrating repository," in *8th IEEE Workshop on Model-Based Development for Computer-Based Systems*, 2011, pp. 284–291.

[34] S. Rochimah, W. Wan Kadir, and A. H. Abdullah, "An evaluation of traceability approaches to support software evolution," in *2nd Intl. Conf. on Advances in Software Engineering*, 2007.

[35] S. Imtiaz, N. Ikram, and S. Imtiaz, "Impact analysis from multiple perspectives: Evaluation of traceability techniques," in *3rd Intl. Conf. on Software Engineering Advances*, 2008, pp. 457–464.

[36] S. A. Bohner, "Extending software change impact analysis into COTS components," in *Annual NASA Goddard Software Engineering Workshop*, 2002, pp. 175–182.

[37] K. Pohl, "PRO-ART: Enabling requirements Pre-traceability," in *2nd Intl. Conf. on Requirements Eng.*, 1996, pp. 76–84.

[38] P. Letelier, "A framework for requirements traceability in UML-based projects," in *1st Intl. Workshop on Traceability in Emerging Forms of SE (TEFSE'02)*, 2002, pp. 32–41.

[39] G. Spanoudakis, A. Zisman, E. Perez-Minana, and P. Krause, "Rule-based generation of requirements traceability relations," *Journal of Systems and Software*, vol. 72, no. 2, pp. 105–127, 2004.

[40] G. Spanoudakis and A. Zisman, "Software traceability: A roadmap," in *Handbook of Software Engineering and Knowledge Engineering*, C. S. K., Ed. River Edge, NJ: World Scientific Publishing Co., 2005, vol. III, pp. 395–428.

[41] W. Jirapanthong and A. Zisman, "Xtraque: traceability for product line systems," *Software and Systems Modeling*, vol. 8, no. 1, pp. 117–144, 2009.

[42] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Detecting and repairing inconsistencies across heterogeneous models," in *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, 2008, pp. 356–364.

[43] "EMFTrace - Sourceforge Project Website." [Online]. Available: https://sourceforge.net/projects/emftrace/

[44] G. A. A. C. Filho, A. Zisman, and G. Spanoudakis, "Traceability approach for i* and UML models," in *Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'03)*, 2003.

[45] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[46] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Journal of Empirical Software Eng.*, vol. 14, pp. 131–164, 2009.

[47] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*, J. Marciniak, Ed. John Wiley & Sons, 1994, pp. 528–532.

[48] S. Hassaine, F. Boughanmi, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "A seismology-inspired approach to study change propagation," in *27th Intl. Conf. on Software Maintenance (ICSM 2011)*, 2011, pp. 53–62.