A Taxonomy of Change Types and its Application in Software Evolution

Steffen Lehnert, Qurat-ul-ann Farooq, Matthias Riebisch
Department of Software Systems / Process Informatics
Ilmenau University of Technology
Ilmenau, Germany
{*steffen.lehnert, qurat-ul-ann.farooq, matthias.riebisch*}@*tu-ilmenau.de*

***Abstract*—Changes play a central role in software evolution, where the types of changes are as multifarious as their consequences. When changing software, impact analysis and regression testing are required to preserve the quality of the system. However, without a consistent classification of types of change operations, a well-founded impact analysis methodology cannot be developed. Existing works which analyze and apply change types are typically limited to a certain aspect of software, e.g. source code or architecture. They also lack a thorough investigation of change types, which lead to duplicated proposals and the absence of a consistent taxonomy. In this paper, we review the usage of change types for impact analysis and regression testing, and illustrate how both activities are affected by different types of changes. Therefore, we outline how existing work deals with different types and granularities of changes. Our main contribution is a generic, graph-based description of changes which distinguishes between atomic and composite change types. We show how existing change types and classifications can be mapped onto our proposed approach and change taxonomy. Finally, we illustrate how our proposed change types can support real developer activities, such as refactorings, impact analysis, and regression testing.**

***Keywords*-software evolution; change types; impact analysis; regression testing; graphs**

I. Introduction

Changes are an essential part of software development and reflect the evolution of software systems. The reasons for changes are as multifarious as their different types. Software has to adapt to new hardware, new middleware, new requirements, and even changing legal circumstances. Changes on the other hand constitute high risks, as they may result in new bugs and decline the quality of software [1]. Frequent changes can lead to an erosion of software architectures and increase the drift between source code and architecture.

Impact analysis can be applied to support evolutionary changes, by improving comprehension, understanding of side effects, and ensuring the completeness of changes [1]. Understanding the different types of changes allows better understanding and reacting on the actual change, for example, when performing maintenance tasks. Furthermore, understanding changes is essential for understanding the evolution of a software system. An assessment of the impact of a certain change requires explicit information about the nature of the change, about the type of the involved artifacts, as well as about dependency relations between the artifacts. This paper deals with the changes as the first part of these requirements. Likewise, analyzing change types can help with identifying test cases which should be rerun during regression testing [2]. Based on impact analysis and regression testing, developers are able to better understand and actually change the software. Additionally, it enables project managers to determine the costs of a proposed change and, if necessary, to take appropriate measures.

We reviewed existing literature, which is concerned with change types. Our review covered the fields of impact analysis, regression testing, refactoring, and reengineering. As a first result, we noticed that most studies lack a comprehensive investigation of change types, including the comparison to related work. We identified a lot of duplicated work in terms of proposed change types and change type taxonomies, which are inconsistent to each other. Therefore, we felt the need for thoroughly comparing existing work, deriving a new taxonomy of changes, and showing how to map existing work onto our developed taxonomy.

Our contribution is fourfold. First, we review how different change types influence impact analysis and regression testing, by examining how approaches deal with the different types. Based on our review of existing literature, we propose a set of atomic change types and show how composite change types can be modeled as sequences of atomic operations. The concept of atomic and composite changes is further embedded in a taxonomy of change types, which is the second contribution. Third, we demonstrate how typical engineering activities, such as refactorings, can be mapped on our atomic and composite change types. Last, we illustrate how rule-based impact analysis and regression testing can benefit from well-defined change types, provided in this paper. The remainder of this paper is organized as follows. We begin with analyzing existing work on change types (Section II), before preparing the ground for our new taxonomy by introducing the concept of atomic and composite changes (Section III). We then introduce our taxonomy for change types (Section IV), illustrate how real changes can be mapped onto our concept (Section V), and further show how rules for impact analysis and regression testing can be modeled using our change types. Finally, Section VI concludes the paper and outlines further work.

II. Change Types in Research and Practice

A. Impact Analysis

In this section, we review studies which have been published in the field of software change impact analysis. We selected studies that introduce change type classifications or list change types as a base of their analysis.

The work of Kung et al. [3] is focused on impact analysis in object oriented class libraries. Thus, they analyze library changes and support the following change types: the addition and deletion of classes, interfaces, methods, variables, method parameters, and control statements. Further, they support type, value, and accessibility changes of data members, and return type changes of methods. However, the authors do not support all types of changes, e.g. the declaration of a method or variable as constant. Also, they do not consider complex operations, such as the extraction of a method or merging two classes.

The change type taxonomy proposed by Li and Offut [4] contains add, delete, and accessibility changes of classes, methods, and attributes, as well as value-change operations of attributes. As in the work of Kung et al. [3], some change types are missing, for example *final*, *static* or *const* modifier changes. Likewise, Li and Offut neglect any complex change operations which are typical for software maintenance and evolution tasks.

Fluri and Gall [5] distinguish between declaration and body part changes in object oriented software. The proposed concept of atomic change types is build on an analysis of elementary tree edit operations (add, delete, and substitute), which are mapped on four atomic types, as defined by Fluri and Gall: *insert*, *delete*, *update*, and *move*. The change types are instantiated for classes, methods, and attributes in Java, which is based on abstract syntax trees. Fluri and Gall perform a thorough investigation of changes on code level; however, they limit their work on atomic change types only. In practice, more complex change operations are likely to occur, e.g. when refactoring a class or a method. Thus, investigating atomic change operations only is not sufficient for supporting developers. The authors further introduce the concept of *change significance*, i.e. they are evaluating change types based on the severity of their impact. So far, this analysis has been conducted on Java source code only, and might not be generalizable to other programming languages or to architectural design models. Finally, the authors analyze change types for their potential of affecting the functionality of a software system, which they refer to as *functionality-preserving*.

Another taxonomy of change types is presented in the work of Feng and Maletic [6], who are concerned with impact analysis in component-based software architectures. The authors propose a classification scheme which distinguishes between atomic changes (add a method, add an interface, remove a method, remove an interface) and composite changes (e.g. "create a new, empty interface and add a method to it"). Their proposed taxonomy is very limited in its applicability, as only the addition and deletion of entities is supported, but no updates of values or properties.

Sun et al. [7] propose a taxonomy for atomic source code changes developed for the Java programming language, which plays a central role in their impact analysis approach. The authors support atomic change operations on classes, methods, and attributes, such as *add*, *delete*, *accessibility modification*, and others. However, the authors neglect method body changes, statement changes, and, similar to approaches discussed above, complex change types.

Co-evolution patterns are a useful source for impact analysis, as co-changing entities are likely to be affected by changes of related entities. Xing and Stroulia [8]–[10] developed an approach to enable the model-based differencing of UML class models, with the overall goal of detecting class co-evolution patterns. The developed UMLDiff tool supports the detection of *add*, *delete*, *move*, *rename*, and *signature changes* of classes, methods, and attributes. The category of signature changes, however, is too coarse-grained, as many different types belong to this category, such as the addition of method parameters or return type changes, as shown in the work of Sun et al. [7]. Some composite change types are also not considered by Xing and Stroulia, for instance the *split* and *merge* operations.

Gupta et al. [11] developed a dynamic impact analysis approach, which is built on a classification of change types. The authors distinguish between four different types of changes: functional changes (changed statements which affect functions), logical changes (control-flow changes), structural changes (the addition or deletion of code entities), and behavioral changes (e.g. change of execution order, change of program entry and exit). However, their classification is ambiguous and not based on real change types. For example, the deletion of a class affects the structure but also the functionality of the system, since the functionality which was implemented by this class was deleted. Therefore, we doubt that this classification scheme is applicable and useful in practice, as the authors did not provide concrete examples.

B. Regression Testing

Störzer et al. [12] propose *JUnit/CIA* as an extension to *JUnit*, which is based on the *Chianti* tool developed by Ren et al. [13], [14]. Both tools depend on the early work of Ryder and Tip [2], who are concerned with impact analysis and regression testing, and introduced sets of atomic change operations. Ryder and Tip, as well as Störzer et al., utilize the semantics of change types to determine affected test cases. The proposed classification of change types is comprised of atomic operations such as the addition of classes.

The majority of studied, model-based regression testing approaches uses *add* and *delete* as primary change types for change identification. The details of these approaches and the change types they use can be found in our previous work on the analysis of regression testing approaches [15]. Some model-based regression testing approaches also use *modify* as another change type. An entity or model element is referred to as modified, if any of its properties is changed [16], [17]. This is similar to the change type *property_update* in our change taxonomy, as presented in Section III-C.

C. Requirements Engineering

A taxonomy for requirements changes has been established and evaluated by McGee and Greer [18], [19]. Their work is focused on classifying the different sources of requirements changes, which the authors refer to as *trigger*. The proposed taxonomy is comprised of three main criteria, namely *Change Domain*, *Trigger*, and *Uncertainty*. However, their taxonomy does not provide criteria to classify the types of changes, as they investigate the sources of change.

D. Other Classification Schemes

Modular design and modularizing software is the focus of the work of Baldwin and Clark [20], who propose a set of atomic operations for the task of modularizing software. The authors distinguish between six distinct developer activities: *splitting* (convert a single-level design into a hierarchical design), *substitution* (replace one design by another), *augmenting* (add a module), *excluding* (remove a module), *inversion* (move a hidden module up in the hierarchy to make it visible to others), and *porting* (a hidden module moves up in the hierarchy and can be used in other contexts). Their proposed set of operations contains a mixture of atomic and composite types. However, the atomic operations are incomplete, as the modification of properties is not covered. The set of composite operations lack the *merge* and the simple *move* operators.

Mens and Buckley [21], [22] established a taxonomy for software changes, which also supports the type of change operation as one classification criterion, which is situated on their *Change support (how)* axis. The authors further distinguish between structural and semantic change types, where structural changes cover the addition, subtraction, and alteration (modifying an existing element, e.g. renaming) of software entities. Semantic changes span semantics-modifying changes, semantics-preserving changes, and restructuring activities. Their taxonomy, therefore, provides a set of atomic operations, but no composite change types. The concept of distinguishing between structural and semantic changes is interesting, however, it suffers from the same problems as the distinction of functionality-preserving changes as introduced by Fluri and Gall [5].

The work of Mäder et al. [23], [24] is aimed at automatically maintaining traceability relations during software evolution. The authors developed and implemented a rule-based concept, which reacts on different change types to update traceability relations among changed entities accordingly. The tool is able to react on elemental operations and *developer activities*, which are comprised of elemental change operations. Elemental change operations span the addition and deletion of elements, as well as property modifications. The set of supported *developer activities* is comprised of *replace*, *merge*, and *split* operations. It is however incomplete, as *swap* and *move* operations are missing.

III. Towards a new Taxonomy of Change Types

This section prepares the ground for our change type taxonomy which is introduced in Section IV. We continue our previous work on change types for regression testing [15], and compare it to the work of Fluri and Gall [5]. We extend their tree-based software representation to the level of labeled graphs, and introduce our concept of atomic and composite changes, which is the central part of our taxonomy. We further illustrate how our concept of labeled graphs can be mapped on the *Eclipse Modeling Framework* (EMF) in Section III-B to demonstrate its applicability in practice.

A. Software - a labeled Graph

We define software as a directed graph of arbitrary artifacts which may contain circles, i.e. $G = (V, E)$. Software artifacts, such as UML diagrams or C++ classes are inserted as nodes (V) and dependencies between them are added as edges (E) to the graph, whereas attributes are added as property labels to the nodes and edges. This kind of representation stands in contrast to the work of Fluri and Gall [5], who treat software as a tree (AST). However, it conforms to most modeling languages, such as UML. The property labels which can be assigned to nodes and edges depend on the domain and the purpose for which the graph shall be used. However, there are two properties which are independent of domain: the *name* and the *type* of an artifact. Thus, each artifact has at least two properties p_j.

$$\forall v_i \in V : v_i = \{p_j | j \in \mathbb{N}, j \geq 2\}$$

Relations which exist between entities can also be enhanced with properties p_l, such as the type of relation (e.g. *ImplementedBy*, *Inherits* or *DefinedBy*).

$$\forall e_k \in E : e_k = (a, b, \{p_l | l \in \mathbb{N}\}),\ where\ a, b \in V$$

This generic graph concept is illustrated by Figure 1, which shows an excerpt of our case study which is introduced in Section V-B. The central node of the graph, the actual *Student Enrollment System*, consists of requirements, architecture, source code, configuration files (not displayed), and other data sources, which are modeled as nodes (radiused boxes). Possible properties of software artifacts, such as data types, visibility information, and other modifiers are all being modeled as property labels (square boxes).

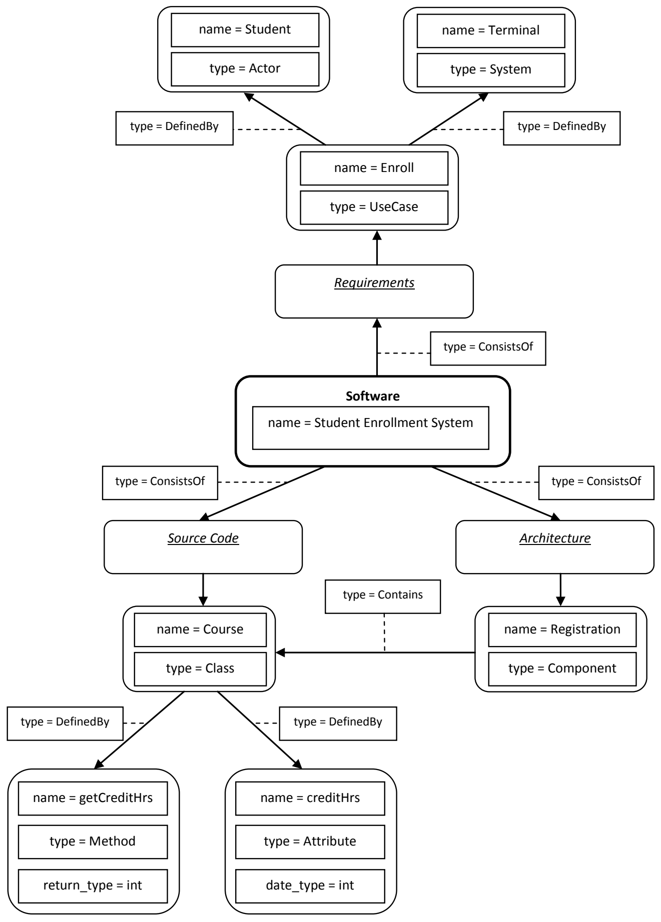

Figure 1. A small excerpt from our Student Enrollment System case study.

B. Mapping to the Eclipse Modeling Framework (EMF)

In the following short section, we want to demonstrate that our concept of generic graphs can be applied to the metamodels described using EMF. This enables us to generalize our approach for the family of languages which are expressed in EMF, such as UML. In the following, we map the elements of the EMF metamodel onto our labeled graph representation. Thus, any instance of a model which is defined in EMF can also be presented as a labeled graph.

1) **Nodes:** EPackage, ECLass, EDataType, EEum, EAnnotation, EOperation, EAttribute, EEnumLiteral.
2) **Edges:** EReference, Inheritance, EAnnotationLink.
3) **Properties:** The attributes of the classes classified as nodes and edges will be mapped to properties.

EMF classes are used to describe the concepts of the language. According to the mappings presented above, they can be mapped onto the nodes in the graph. EMF classes can have attributes and relations to other classes. The classes *EReference*, *Inheritance*, and *EAnnotationLink* of the EMF metamodel represent the relations in a model. We map these classes onto edges in our metamodel, as presented in the mappings above. The attributes of the *EClasses*, which have been classified as nodes and edges, will be mapped onto properties in our labeled graph, as shown by Figure 1. Using these mappings, we are able to represent all the elements of EMF based models as a labeled graph.

C. Atomic Changes

After introducing the labeled graph representation in Section III-A, we are now able to introduce our set of atomic change operations, OP_{atomic}. This set is similar to the one proposed by Fluri and Gall [5], and will be used to model composite change types in Section III-D.

$$OP_{\text{atomic}} := \{add_{\text{node}}, delete_{\text{node}}, add_{\text{edge}}, delete_{\text{edge}}, \\ update\_property\}$$

In contrast to Fluri and Gall, we do not consider *move* as an atomic operation, since it can be modeled by *delete* and *add* operations. We further distinguish between the *addition* of nodes and edges, which also applies for the *delete* operation. A detailed comparison to the work of Fluri and Gall [5] is depicted by Figure 2, and a mapping of change types is presented by Table II in the appendix.

D. Composite Changes

Our proposed composite change types consist of sequences of atomic operations and are based on previous research on regression testing [15] and traceability maintenance [23]. As previously stated, we consider *move* as an composite change type, since it can be modeled by a sequence of *add* and *delete* operations. Proposed composite change types also share some similarities with the refactoring activities proposed by Baldwin and Clark [20]. Therefore, the set of composite operations $OP_{\text{composite}}$ is defined as follows.

$$OP_{\text{composite}} := \{move, replace, split, merge, swap\}$$

- Move - move one sub-graph to another node.
- Replace - replace one sub-graph by another sub-graph.
- Split - split one sub-graph into several sub-graphs.
- Merge - merge several sub-graphs into one.
- Swap - exchange two sub-graphs.

The following subsections will explain each composite change type and how it can be modeled as a sequence of atomic operations. The function $P(x)$ denotes the "direct parent" of node x, i.e. one of its predecessors which is related to x via a relation of the type *DefinedBy*, *ConsistsOf* or similar. For example, a method *m* can have more than one predecessor, i.e. one class *C* it belongs to and several other artifacts, which call this method (e.g. related via *Calls*). Thus, we consider *C* as the "parent" of *m*, and *m* as the "child" of *C* likewise.

1) Move-operation: We support two versions of the move operation. First, the developer can move an entire sub-graph x to another node y, e.g. when moving a class to another package (all methods and attributes are also moved).

$$move(x,y) := delete_{\text{edge}}(x, P(x)), add_{\text{edge}}(x,y).$$

On the other hand, a developer might want to move a node x to another node y only, leaving potential child nodes x_i in place, e.g. when moving an attribute up to the parent class.

$$move'(x,y) := \bigwedge_{i=0}^{n} move(x_i, P(x)), move(x,y).$$

Depending on the context, the *move* operation may also be modeled as a sequence of $delete_{\text{node}}$ and add_{node} operations. This might occur in the context of version control repositories, when extracting change operations from log files, such as CVS or SVN.

2) Replace-operation: In a similar fashion, a developer can replace the entire sub-graph x by another sub-graph y, e.g. when overloading a method which is inherited from the superclass.

$$replace(x,y) := delete_{\text{node}}(x), move(y, P(x)).$$

The developer may also replace a node x by node y only, i.e. leaving all the child nodes x_i of x in place.

$$replace'(x,y) := \bigwedge_{i=0}^{n} move(x_i, y), replace(x,y).$$

3) Split-operation: The *split*-operation creates a set of *n* nodes $(n \in \mathbb{N}, n \geq 2)$ which are of the same type as x, and moves all child elements of x to the respective new node x'_i. An example for this operation is the extraction of a class from another class. A new class is created and all methods and attributes which should be extracted are moved to the new class. A tuple (s_a, d_b) denotes, that the s_a*-th* sub-graph of the node x should be moved to the d_b*-th* sub-graph of the resulting set x'.

$$split(x, n, (s_0, d_0) \ldots (s_m, d_m)) := \bigwedge_{i=0}^{n} add_{\text{node}}(x'_i, P(x)),$$
$$\bigwedge_{j=0}^{m} move(y_{s_j}, x'_{d_j})),\ where\ x = P(y_{s_j}).$$

4) Merge-operation: The inverse operation to *split* is *merge*, which bundles n entities $(n \in \mathbb{N}, n \geq 2)$ of the same type into one, where y_{i_j} is the j-*th* sub-graph of x_i.

$$merge(x_0 \ldots x_n) := \bigwedge_{i=1}^{n} (\bigwedge_{j=0}^{m} move(y_{i_j}, x_0)).$$

5) Swap-operation: Swapping allows exchanging two entities by another.

$$swap(x,y) := move(x, P(y)), move(y, P(x)).$$

We also support exchanging nodes only, which attaches the child nodes of x, x_i, to y, and vice versa.

$$swap'(x,y) := \bigwedge_{i=0}^{n} move(x_i, y), \bigwedge_{j=0}^{m} move(y_j, x),$$
$$swap(x,y).$$

	Fluri and Gall	Our approach
1. Representation		
Type	Tree	Graph
Elements	Node	Node
Relations	Edge	Edge
Attributes	Leaf	Property-label
2. Atomic Change Types		
Add	Insert	Add_{node}, $\text{Add}_{\text{relation}}$
Delete	Delete	$\text{Delete}_{\text{node}}$, $\text{Delete}_{\text{relation}}$
Update	Update	Update_Property
3. Composite Change Types		
Move	Move	Move
Replace	-	Replace
Split	-	Split
Merge	-	Merge
Swap	-	Swap

Figure 2. A comparison between Fluri and Gall [5] and our approach

Further combinations of atomic and composite changes presented here are possible. A developer for example might want to add two or more methods to a class, which corresponds to a sequence of atomic *add* operations, and is accompanied by a series of *property_update* operations. Inverting the hierarchy between two entities can also be realized, using a series of composite and atomic changes. If the inheritance hierarchy between two classes shall be inverted, one would *swap* both nodes (i.e. the classes) and then resolve attributes and methods by either *adding*, *deleting* or *moving* them between both classes. Likewise, the *inversion* and *porting* operations of Baldwin and Clark [20] can be modeled using our approach.

IV. A Taxonomy of Change Types

In this section, we introduce a set of classification criteria which contribute to our taxonomy. We distinguish between *generic* and *concrete* change types by the criterion *abstraction level*, as change types presented so far were generic ones, i.e. they have to be instantiated for the concrete case. Secondly, we distinguish between *atomic* and *composite* changes. We refer to this criterion as *composition type*. For example, "moving an entity" is an abstract, composite change. In contrast, "adding a class X to package Y" is a concrete, atomic operation. Our third criterion *type of* *operation* reflects the either atomic or composite type of change, e.g. *add* or *move*. The forth criterion *scope of change* is required to associate a possible change type with the kind of software artifacts it can be applied on, e.g. object oriented source code or architectural models. Therefore, our taxonomy is established as illustrated by Figure 3.

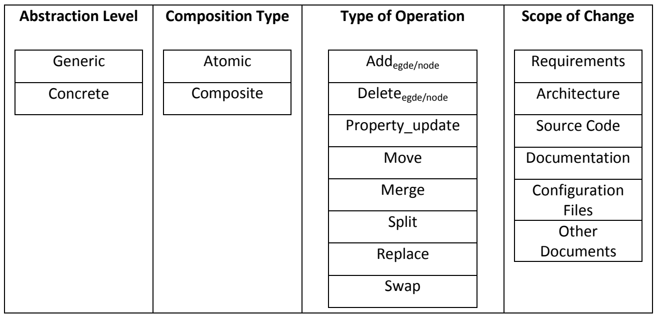

Figure 3. A taxonomy for change types.

Before we demonstrate how our taxonomy can be used in practice, we want to exemplify the classification of change types using the taxonomy. The change "set the return type of method X to *integer*" would be classified as follows.

- Abstraction level: concrete.
- Composition type: atomic.
- Type of operation: property_update.
- Scope of change: Architecture, Source Code.

V. Applications in Practice

In this section we show how our concept of atomic and composite changes can be used to model concrete changes and refactoring activities, and how those operations are classified according to our taxonomy. Furthermore, we illustrate how our taxonomy and its well-defined change types serve as preconditions for rule-based impact analysis and regression testing.

A. Concrete Change Types

So far, our taxonomy and the concept of atomic and composite changes are of a rather abstract nature. In the following section, we list a set of real changes. We begin with listing atomic operations, where we focus on property updates, as this allows us to demonstrate a variety of types in contrast to *add* and *delete*.

Property_Update. The amount of possible property updates is vast, thus we will limit our listing to a few types which may occur in OOP and architectural design.

- Declare a class as *abstract*.
- Change the visibility of a class, method or attribute to either *public*, *protected* or *private*.
- Rename an artifact, e.g. a class.
- Change the type of an attribute, the return type of a method or the type of a method parameter.
- Change the modifiers of code artifacts, e.g. declare them as *virtual*, *const*, *static* or *final*.

The following lists a series of composite change types, where their classification is summarized by Table I.

Move. Fowler [25] proposes moving of methods as a refactoring step, where Gorp et al. [26] use move operations to pull methods up into a superclass.

Split. Sunyé et al. [27] extract sub-states from UML state machines by splitting them into a set of states.

Merge. Merging transitions in UML state machines is illustrated by Sunyé et al. [27], where Boger et al. [28] also support the merge of states.

Swap. Swapping may occur whenever two entities are replaced by another. For example, two program statements may be swapped, if their order of execution shall be changed.

Replace. The extraction of a method, which replaces statements by method calls, is proposed by Gorp et al. [26]. Other examples can be found in the work of Sunyé et al. [27], who replace UML state machine transitions, and in the work of Westhuizen and Hoek [29] who replace architectural elements during architectural evolution.

Author	Abstraction Level	Composition Type	Type of Operation	Scope of Change
Fowler [25]	concrete	composite	Move	Code
Gorp et al. [26]	concrete concrete	composite composite	Move Replace	Code Code
Sunyé et al. [27]	concrete concrete concrete	composite composite composite	Split Replace Merge	Arch. Arch. Arch.
Boger et al. [28]	concrete	composite	Merge	Arch.
Westhuizen and Hoek [29]	concrete	composite	Replace	Arch.

Table I
A CLASSIFICATION OF THE PRESENTED CHANGES.

B. Example Application

Before we introduce our approach for impact detection rules and show how to utilize them for regression testing, we introduce our example application "Student Enrollment System" (see also Figure 1). Figure 4 presents an excerpt of the UML class diagram of the "Student Enrollment System". The complete case study and the test generation methodology for the case study are discussed in our previous work [15]. Every class of the "Student Enrollment System" which has a state dependent behavior is defined by a corresponding state machine. Figure 5 represents one example sequence of states and transitions from the *Student*-state machine.

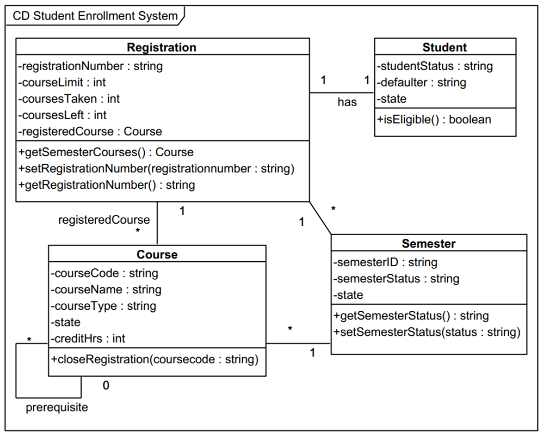

Figure 4. Excerpt of our student enrollment case study

Transition T5 in Figure 5 shows an example, where the value of the *status* variable is assigned to *enrolled*, and the state of the system changes from *BeingEnrolled* to *Enrolled*.

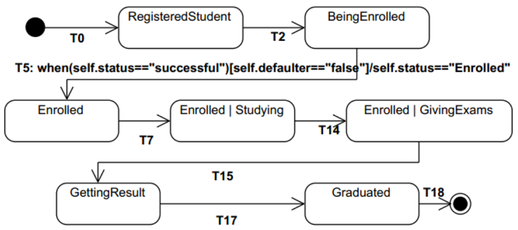

Figure 5. An example sequence from the Student state machine

C. Rule-based Impact Analysis

In previous work [30], [31] we analyzed how impact analysis is achieved, and which methodologies are applied by researchers. We identified a set of ten recurring methods [30], for example call graph analysis (e.g. Korpi and Koskinen [32], Vidács et al. [33]), history mining (e.g. Zimmermann et al. [34], Hassan and Holt [35]) or information retrieval (e.g. Poshyvanyk et al. [36], Antoniol et al. [37]).

Our research hypothesis, however, is that the impact of a change depends on four different factors which can be determined by a developer who is changing the software. First, the type of change is important, since renaming a class has a lower impact in modern IDEs than changing the return type of a method for instance. This is also the part where our taxonomy of change types comes to play, as it provides a set of generic operations on which real changes can be mapped upon. The type of the dependency relation between the changed artifact and a possibly impacted artifact is the second factor, since the relation-type determines if and how a change "ripples" across the artifacts. Last, we consider the types of the changed and related artifact as factors, as they influence the change propagation. We propose a technique which is based on impact rules and belongs to the category of "Explicit Rules", according to our taxonomy of impact analysis [30]. Using these factors, it is possible to create a set of rules to react on changes and to propose possibly impacted artifacts to the developer, which we will explain by example. Therefore, assume we want to add the method *getCourse* to class *Registration*, to enable querying for a certain course using its identifier. Traditional dependency analysis would propose class *Student* as impacted, since it contains an instance of *Registration*. History mining would propose *Student* as impacted, if both classes were frequently committed together, which might be coincidence or due to habits of the developer(s). Finally, a call graph analysis would also yield *Student* as impacted, since the class calls a method of a changed class. However, no adaptation is required. There would be an impact in class *Student*, if we would change the attribute type of *registrationNumber*, since this also requires to change the return type of *getRegistrationNumber*. Thus, a set of rules can be created (Listing 1-2), where the change types *ChangeDataType* and *ChangeReturnType* could be mapped on the *property_update*-operation of our taxonomy.

```
begin rule(entity x, y; change c)
   report x as affected when
      x.type == method
      y.type == attribute
      c.type == ChangeDataType
      c.target == y
      x.relationTo(y) == ReturnsValue
end
```

Listing 1. A rule to react on attribute type changes

```
begin rule(entity x, y; change c)
   report x as affected when
      x.type == attribute OR variable
      y.type == method
      c.type == ChangeReturnType
      c.target == y
      x.relationTo(y) == AssignsValue
end
```

Listing 2. A rule to react on method return type changes

We are currently implementing a set of impact rules and a prototype tool to perform a first case study, in order to gauge the feasibility and precision of the proposed approach. Section V-E discusses this in more detail.

D. Rules for Regression Testing

Before we discuss the rules to identify impacted test cases, we briefly discuss the relations of the state machine with the test cases. A test path is a sequence of transitions, which is derived from the state machine and is used to test the class which corresponds to the state machine. The test suite of the *Student*-class contains 58 test paths in total. The other details, such as the test derivation methodology, are discussed in our previous work [15]. As outlined in Section V-B, a transition can refer to the attributes and operations of classes. This means, a change in an attribute or method of a class can affect the test cases as well. If such a change occurs, the affected test cases should be identified to perform regression testing and test suite maintenance.

To explain the effect of changes on the tests, we refer to the example of attribute type changes, discussed in Section V-C. In case of attribute type changes, the methods which are returning these attributes are marked as affected, as shown in Listing 1. In case of a method being affected, every transition which uses this method in its events, guards, or actions should also be marked as affected. As a result, the test sequences which contain this transition should also be considered as affected. These test sequences need to be rerun to reveal the regression faults. Listings 3 and 4 present the rules for identifying the affected transitions and their corresponding test paths.

```
begin rule(entity t, m)
   report t as affected when
      m.type == method
      t.type == transition
      t.relationTo(m) == uses
      m.isAffected()
end
```

Listing 3. A rule for affected transitions

```
begin rule(entity t, p)
   report p as reusable when
      t.type == transition
      p.type == path
      p.relationTo(t) == is_derived_by
      t.isAffected()
end
```

Listing 4. A rule for affected test paths

According to the rule presented in Listing 3, a transition is reported as affected if it relates to the method *m* with the relation type *uses*, and the method *m* is an already affected method (obtained by applying the rule in the Listing 1). The other rule in Listing 4 checks, if the test path and the transition satisfy the relation *is_derived_by*, and if the transition is already affected. If both conditions are met, the test path is classified as affected. More complex rules to maintain the test suites can be derived by considering other complex change types. For example, if we *split* a class in to *n* classes, a lot of existing test paths become invalid and should be reported as, for example, *obsolete*.

E. Implementation and Evaluation

We are currently implementing our concept of impact rules in a prototype tool, to assess their applicability and performance. Therefore, we extend our prototype CASE tool *EMFTrace* [38], which has initially been developed for automated traceability detection among models of various modeling languages. The tool provides a rule engine, is based on a model repository, and supports a variety of modeling languages (UML, OWL, BPMN, URN, Feature Models, and factor tables & issues cards [39]).

We are planning a two-pronged evaluation strategy, consisting of an initial case study to assess the feasibility of our approach and a detailed evaluation regarding its performance. We are following the guidelines for case study research as established by Runeson and Höst [40] and the *Goal Question Metric* (GQM) introduced by Basili et al. [41] to ensure the quality of its results. We are currently developing a case study protocol for the initial study, which we plan to have peer-reviewed. We plan to use the course management example presented in Section V-B, and a series of more complex, real systems for evaluating our approach. We are currently investigating eight different systems, including a robot control software and a open source render engine according to the criteria defined in our protocol.

The actual evaluation of our impact rules will be achieved by using the metrics of recall and precision, which are based on the *actual impact set* (AIS) and the *estimated impact set* (EIS). The AIS contains all entities which are affected by a change, where the EIS contains all entities which are proposed to be affected. Precision and recall are then determined as shown in [30]. The AIS is obtained by developers, who assess the impact of a proposed change in terms of artifacts which require rework. The same change is then fed into the tool to compute the EIS. Finally, both sets will be compared to obtain the figures for precision and recall. Based on those numbers, we will then be able to compare our approach to existing work in the fields of impact analysis and regression testing.

VI. Conclusion

We analyzed approaches proposed for impact analysis, regression testing, and other tasks which are based on different classifications of change operations. A review of the proposed change type classifications revealed inconsistent, partly duplicated, and incomplete taxonomies which are incompatible to each other.

We extracted common change types from studied literature and embedded them in a new taxonomy, which is based on the distinction between atomic and composite changes. We illustrated how existing work can be mapped on our taxonomy, and how real change operations can be classified according to our criteria. We further outlined how rules for impact analysis and regression testing can benefit from a well-founded taxonomy of change types.

Future work is aimed at evaluating dependency relations between artifacts, to determine the impact of changes on refactorings and regression testing, using our impact rules. The required impact rules, which are currently under development, shall be evaluated and refined in a preceding case study, to assess the feasibility of the approach. We are further planning a thorough evaluation to compare the performance of impact rules to other techniques. Also, we plan to perform a more thorough investigation on the role of change types for refactoring activities and for systems in different domains.

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] B. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01)*, Snowbird, Utah, USA, June 2001, pp. 46–53.

[3] D. Kung, J. Gao, P. Hsia, and F. Wen, "Change impact identification in object oriented software maintenance," in *International Conference on Software Maintenance*, Victoria, BC, Canada, September 1994, pp. 202–211.

[4] L. Li and A. J. Offutt, "Algorithmic analysis of the impact of changes on object-oriented software," in *Proceedings of the International Conference on Software Maintenance*, Monterey, CA , USA, November 1996, pp. 171–184.

[5] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proceeding of the 14th IEEE International Conference on Program Comprehension (ICPC 2006)*, Athens, 2006, pp. 35–45.

[6] T. Feng and J. I. Maletic, "Applying dynamic change impact analysis in component-based architecture design," in *Proceeding of the Seventh International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2006)*, Las Vegas, Nevada, USA, June 2006, pp. 43–48.

[7] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, "Change impact analysis based on a taxonomy of change types," in *Proceedings of the IEEE 34th Annual Computer Software and Applications Conference*, Seoul, Korea (South), July 2010, pp. 373–382.

[8] Z. Xing and E. Stroulia, "Data-mining in support of detecting class co-evolution," in *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE'04)*, June 2004, pp. 123–128.

[9] ——, "Understanding class evolution in object-oriented software," in *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)*, June 2004, pp. 34–43.

[10] ——, "UMLDiff: An algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05)*, Long Beach, California, USA, November 2005, pp. 54–65.

[11] C. Gupta, Y. Singh, and D. S. Chauhan, "A dynamic approach to estimate change impact using type of change propagation," *Journal of Information Processing Systems*, vol. 6, no. 4, pp. 597–608, December 2010.

[12] M. Störzer, B. G. Ryder, X. Ren, and F. Tip, "Finding failure-inducing changes in Java programs using change classification," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, Portland, Oregon, USA, 2006, pp. 57–68.

[13] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, and J. Dolby, "Chianti: A prototype change impact analysis tool for Java," Rutgers University, Department of Computer Science, Tech. Rep. DCS-TR-533, September 2003.

[14] X. Ren, F. Shah, B. Tip, and O. Chesley, "Chianti: A tool for change impact analysis of Java programs," in *Proceedings of the 19th annual ACM SIG-PLAN Conference on Object-oriented programming, systems, languages, and applications (OOPSLA '04)*, Vancouver, BC, Canada, 2004, pp. 432–448.

[15] Q.-u.-a. Farooq and M. Riebisch, *Model-Based Regression Testing: Process, Challenges and Approaches*. IGI Global, 2011, ch. 10, pp. 254–297.

[16] Q.-u.-a. Farooq, M. Z. Z. Iqbal, Z. I. Malik, and A. Nadeem, "An approach for selective state machine based regression testing," in *Proceedings of the 3rd international workshop on Advances in model-based testing*, ser. A-MOST '07. New York, NY, USA: ACM, 2007, pp. 44–52. [Online]. Available: http://doi.acm.org/10.1145/1291535.1291540

[17] L. Briand, Y. Labiche, K. Buist, and G. Soccar, "Automating impact analysis and regression test selection based on UML designs," in *Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM'02)*, Montreal, Quebec, Canada, October 2002, pp. 252–261.

[18] S. McGee and D. Greer, "A software requirements change source taxonomy," in *Proceedings of the Fourth International Conference on Software Engineering Advances (ICSEA '09)*, Porto, September 2009, pp. 51–58.

[19] ——, "Software requirements change taxonomy: Evaluation by case study," in *Proceedings of the 19th IEEE International Requirements Engineering Conference (RE 2011)*, Trento, September 2011, pp. 25–34.

[20] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*. Cambridge, MA, USA: MIT Press, 2000.

[21] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, pp. 309–332, September 2005.

[22] T. Mens, J. Buckley, M. Zenger, and A. Rashid, "Towards a taxonomy of software evolution," in *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution*, Warsaw, Poland, April 2003.

[23] P. Mäder, M. Riebisch, and I. Philippow, "Traceability for managing evolutionary change," in *Proceedings of the 15th International Conference on Software Engineering and Data Engineering (SEDE-2006)*, Los Angeles, California, USA, July 2006, pp. 1–8.

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

[25] M. Fowler, *Refactoring: Improving the design of existing code*. Amsterdam: Addison Wesley, Longman, Inc., 1999.

[26] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards automating source-consistent uml refactorings," *Lecture Notes in Computer Science*, vol. 2863, pp. 144–158, 2003.

[27] G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel, “Refactoring UML models,” *Lecture Notes in Computer Science*, vol. 2185, pp. 134–148, 2001.

[28] M. Boger, T. Sturm, and P. Fragemann, “Refactoring browser for UML,” *Lecture Notes in Computer Science*, vol. 2591, pp. 366–377, 2003.

[29] C. van der Westhuizen and A. van der Hoek, “Understanding and propagating architectural changes,” in *Third Working IEEE/IFIP Conference on Software Architecture*, 2002, pp. 95–109.

[30] S. Lehnert, “A taxonomy for software change impact analysis,” in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution (IWPSE-EVOL 2011)*. Szeged, Hungary: ACM, September 2011, pp. 41–50.

[31] ——, “A review of software change impact analysis,” Ilmenau University of Technology, Department of Software Systems / Process Informatics, Tech. Rep., December 2011.

[32] J. Korpi and J. Koskinen, “Supporting impact analysis by program dependence graph based forward slicing,” in *Advances and Innovations in Systems, Computing Sciences and Software Engineering*, K. Elleithy, Ed. Springer Netherlands, 2007, pp. 197–202.

[33] L. Vidács, A. Beszédes, and R. Ferenc, “Macro impact analysis using macro slicing,” in *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT ’07)*, 2007, pp. 230–235.

[34] 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.

[35] A. E. Hassan and R. C. Holt, “Predicting change propagation in software systems,” in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, September 2004, pp. 284–293.

[36] 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.

[37] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, “Identifying the starting impact set of a maintenance request: A case study,” in *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, Zurich, Switzerland, February 2000, pp. 227–230.

[38] S. Bode, S. Lehnert, and M. Riebisch, “Comprehensive model integration for dependency identification with EMFTrace,” in *Joint Proceedings of the First International Workshop on Model-Driven Software Migration (MDSM 2011) and the Fifth International Workshop on Software Quality and Maintainability (SQM 2011)*. Oldenburg, Germany: CEUR, March 2011, pp. 17–20.

[39] C. Hofmeister, R. Nord, and D. Soni, “Global analysis: moving from software requirements specification to structural views of the software architecture,” *IEE Proceedings - Software*, vol. 152, no. 4, pp. 187–197, Aug. 2005.

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

[41] 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.

VII. Appendix

Fluri and Gall [5]	Our approach
Additional Object State	$add_{\text{node}}(x, y)$, $update\_property(x, ”type”, ”attribute”)$
Condition Expression Change	$update\_property(x, ”expression”, ”new”)$
Decreasing Statement Delete	$delete_{\text{node}}(x, y)$
Decreasing Statement Parent Change	$move(x, y)$
Else-Part Insert	$add_{\text{node}}(x, y)$
Else-Part Delete	$delete_{\text{node}}(x, y)$
Increasing Statement Insert	$add_{\text{node}}(x, y)$
Increasing Statement Parent Change	$move(x, y)$
Removed Function	$delete_{\text{node}}(x, y)$
Removed Object State	$delete_{\text{node}}(x, y)$
Statement Delete	$delete_{\text{node}}(x, y)$
Statement Insert	$add_{\text{node}}(x, y)$, $update\_property(x, ”type”, ”statement”)$
Statement Ordering Change	$swap(x, y)$
Statement Parent Change	$move(x, y)$
Statement Update	$update\_property(x, ”value”, ”new\_value”)$
Class Renaming	$update\_property(x, ”name”, ”new\_name”)$
Decreasing Accessibility Change	$update\_property(x, ”visibility”, ”value”)$, where $value$ either equals *private* or *protected*
Increasing Accessibility Change	$update\_property(x, ”visibility”, ”value”)$, where $value$ either equals *public* or *protected*
Attribute Type Change	$update\_property(x, ”data\_type”, ”new”)$
Attribute Renaming	$update\_property(x, ”name”, ”new\_name”)$
Final Modifier Insert	$update\_property(x, ”final”, ”true”)$
Final Modifier Delete	$update\_property(x, ”final”, ”false”)$
Method Renaming	$update\_property(x, ”name”, ”new\_name”)$
Parameter Delete	$delete_{\text{node}}(x, y)$
Parameter Insert	$add_{\text{node}}(x, y)$, $update\_property(x, ”type”, ”parameter”)$
Parameter Ordering Change	$swap(x, y)$
Param. Type Change	$update\_property(x, ”data\_type”, ”new”)$
Parameter Renaming	$update\_property(x, ”name”, ”new\_name”)$
Parent Class Delete	$delete_{\text{node}}(x, y)$
Parent Class Insert	$add_{\text{node}}(x, y),\ add_{\text{relation}}(x, y, z)$, $update\_property(z, ”type”, ”IsA”)$
Parent Class Update	$delete_{\text{relation}}(x, y),\ add_{\text{relation}}(x', y, z)$, $update\_property(z, ”type”, ”IsA”)$
Return Type Delete	$update\_property(x, ”return\_type”, ”void”)$
Return Type Insert	$update\_property(x, ”return\_type”, ”new”)$
Return Type Update	$update\_property(x, ”return\_type”, ”new”)$

Table II
Mapping between Fluri and Gall [5] and our approach