Steffen Lehnert

A review of software change impact analysis

Technische Universität Ilmenau
Ilmenau, 2011

URN: urn:nbn:de:gbv:ilm1-2011200618

A Review of Software Change Impact Analysis

Steffen Lehnert
Department of Software Systems / Process Informatics
Ilmenau University of Technology
Ilmenau, Germany
steffen.lehnert@tu-ilmenau.de

***Abstract*—Change impact analysis is required for constantly evolving systems to support the comprehension, implementation, and evaluation of changes. A lot of research effort has been spent on this subject over the last twenty years, and many approaches were published likewise. However, there has not been an extensive attempt made to summarize and review published approaches as a base for further research in the area. Therefore, we present the results of a comprehensive investigation of software change impact analysis, which is based on a literature review and a taxonomy for impact analysis. The contribution of this review is threefold. First, approaches proposed for impact analysis are explained regarding their motivation and methodology. They are further classified according to the criteria of the taxonomy to enable the comparison and evaluation of approaches proposed in literature. We perform an evaluation of our taxonomy regarding the coverage of its classification criteria in studied literature, which is the second contribution. Last, we address and discuss yet unsolved problems, research areas, and challenges of impact analysis, which were discovered by our review to illustrate possible directions for further research.**

I. INTRODUCTION

Performing impact analysis is an important step when changing or maintaining software, especially in incremental processes [1]. It allows to judge the amount of work required to implement a change [2], proposes software artifacts which should be changed [3], and helps to identify test cases which should be re-executed to ensure that the change was implemented correctly [4]. Impact analysis also enables developers and project leaders to ask ”what if...?” questions, and to simulate alternative scenarios without having to implement them.

However, maintenance is considered to be the most expensive [5] and long-lasting phase in the lifecycle of most software systems, where more than 50% of all maintenance costs arise from changing software [6]. Therefore, every development or analysis step which can be automated can save a lot of time and money. On the other hand, partial implemented changes present high risks. They are likely to cause unintended side effects, introduce new bugs, and lead to more instability, rather than improving the software. Today’s evolutionary development and frequent changes demand for changing software, and change has become a daily routine for architects, programmers, and project leaders

The challenges of impact analysis have been addressed for many years and date back to the 1970s when maintaining software and software evolution started to attract the interest of researchers in software engineering. Bohner and Arnold [2], [3], [7], [8] investigated the foundations of impact analysis, and provided the following definition of the term *impact analysis*, which has been adapted by most researchers:

> ”Identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change [2]”.

Today, there are several hundred studies which are concerned with impact analysis. Thus, we felt the need to structure the field by establishing a taxonomy for impact analysis in previous work [9], and to review published literature, which is the aim of this paper.

A. Scope and Contribution of the Review

As in our previous work, we limit our focus on studies describing automated and semi-automated impact analysis approaches. Several experiments have shown that intuition, experience, and skill alone are not sufficient to capture all impacts in complex software, which is why tool and method support is required. Tóth et al. [10] conducted an experiment to compare the results of manual impact analysis to results achieved by several automated approaches, and they concluded by strengthening the need for solid tool support. Likewise, an earlier experiment of Lindvall [11] implies similar results, as the author has shown that developers are not able to determine the impacts of a change in a reliable way. The results of his experiment illustrated the gap between what developers assume to be impacted and what is really impacted. Lindvall quantified this difference as 60% in his experiment.

The focus of this review is not on organizational aspects of impact analysis or the definition of concepts. However, we also reviewed such literature to structure the field of impact analysis, and to develop a classification scheme for technical approaches [9]. This study reviews the following kinds of literature. Articles published at conferences and workshops, articles published in books and journals, technical reports, as well as master and PhD thesis which have been published between 1991 and 2011. We reviewed studies which focus on developing or improving impact analysis approaches, studies which describe experiments to assess the effectiveness of impact analysis techniques, and studies which compare different techniques. We do not consider manual approaches such as reviews and audits, as explained in our previous work [9].

The main contribution of this work is a comprehensive review of approximately 150 studies, which propose or improve impact analysis approaches. We name their motivation, describe the proposed methodology, and classify each approach according to the criteria of our taxonomy [9]. As a byproduct of this classification, we evaluate our taxonomy by checking the coverage of its criteria. This allows us to draw conclusions about its usefulness in practice. The third contribution is the identification of future research work, based on the review of current studies. The following listing summarizes the contribution of this article:

1) A comprehensive review, including
    a) the summary of 150 approaches,
    b) the classification of 150 approaches.
2) Evaluation of the taxonomy and its criteria.
3) Identification of future research areas.

B. Organization of the Paper

This short section explains the structure of the article. The goals of section II are to provide an overview of existing reviews and to introduce the classification criteria, which have been developed in [9] and are used to classify studied literature. The main part of this article, section III, is concerned with analyzing and classifying impact analysis approaches. This section is accompanied by Table II at the end of this article, which contains the results of the actual classification. Section III is further divided according to the different scopes of impact analysis which are explained in section II-B, to allow for easy navigation. Finally, section IV reports and summarizes major results of the review, according to the goals of this article.

II. TOWARDS A REVIEW

In this section, we investigate and evaluate existing attempts to review the field of impact analysis. We analyze studies which are aimed at evaluating approaches, comparing tools, and analyzing the role of certain concepts for impact analysis. Secondly, this section explains the different scopes of impact analysis. Finally, a short summary of our classification criteria is given, which will be used to classify approaches in section III.

A. Previous and related Studies

One of the first attempts to review and structure the field of impact analysis were made by Arnold and Bohner back in 1993 [7]. They established a framework for classifying and comparing approaches which we discussed in [9] according to our goals. Arnold and Bohner reviewed five rather different studies and classified them according to their criteria.

Orso et al. [12] and Breech et al. [13] compared several source code based impact analysis techniques regarding their performance. Orso et al. [12] analyzed two dynamic impact analysis algorithms called *PathImpact* and *CoverageImpact*, and focused their analysis on achieved precision, memory consumption, and execution time. Breech et al. [13] analyzed and compared two dynamic (*PathImpact*, *CoverageImpact*) and two online impact analysis algorithms (*InstrumentedPathImpact*, *DynamicCompilation*). They evaluated the performance of the algorithms according to their scalability, precision, recall, memory consumption, and execution time. However, both studies examine rather similar approaches and do not provide an overview of the wide field of impact analysis.

In a similar fashion, Hattori et al. [14], [15] examined several impact analysis algorithms which they implemented in their *Impala* tool. The authors measured the performance of the algorithms using the information retrieval metrics *precision* and *recall*, and compared their results against results achieved with the *DesginWizard* tool.

A review of Bohner [3] is concerned with the question where and how impact analysis is applied during the development and maintenance process. The author proposes a series of models, which assist with understanding and managing impact detection, change modeling, change implementation, testing, and related tasks. His focus is therefore not on reviewing specific approaches, but on illustrating the integration of impact analysis into the development process.

De Lucia et al. [16] analyzed the role of traceability relations in impact analysis, as they allow to connect different software artifacts and to analyze impact relations between them. The interconnection of different types of artifacts for analytical purpose is important, as changes do not only occur in code and their affects are not limited to code either. Several techniques for establishing traceability links are explored in their work. However, they do not analyze and compare specific impact analysis studies.

A study of different scope was conducted by Robbes et al. [17] who reported on the trend to evaluate impact analysis approaches with development data stored in versioning systems such as CVS or SVN. However, their analysis has shown that these sources cannot always provide accurate and reliable data for an objective evaluation of approaches. Thus, they proposed a benchmark and evaluation procedure for impact analysis approaches to overcome this limitation.

A review of Kagdi et al. [18]–[20] on the topic of Mining Software Repositories (MSR) explores impact analysis as one possible application of MSR. This review is valuable for the impact analysis community, as many studies are examined which apply MSR techniques for change prediction and the detection of co-change patterns which can be utilized for impact analysis.

Kilpinen [21] examined the practical applicability of impact analysis approaches in her PhD thesis. The author identified a gap between approaches proposed for impact analysis and their application in practice. However, most authors neglect this fact and thereby limiting the applicability of their approach. In contrast, experiments and case studies conducted within the scope of her thesis have shown a sharp decrease in required design work when marginally improving the integration of impact analysis into the development process.

The conclusion of this section is as follows. All reviewed studies have a very different focus, e.g. they analyzed techniques and algorithms for impact analysis or the integration of impact analysis into the development process. So far, there is no comprehensive review which explains and classifies research approaches which have been proposed for the task of impact analysis. In contrast, we identified roughly 150 studies which are concerned with impact analysis. Thus, ordering them according to well-defined criteria is an important and necessary task to understand impact analysis.

B. Scopes of Interest

We identified three different scopes of impact analysis approaches [9], as most approaches either analyze source code, formal models or miscellaneous artifacts. Source code analysis can either be static, dynamic or online. Formal models are further divided into architectural and requirements models, to reflect the earlier phases of software development, i.e. requirements capturing and architectural reasoning. Miscellaneous artifacts span a wide range of documents and data sources, such as documentation, bug trackers, and configuration files. Figure 1 illustrates the scopes. The following five subsections outline each scope in more detail, to explain why we have chosen such a distinction.

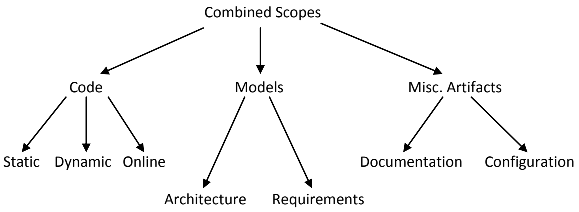

Fig. 1. Scopes of impact analysis

1) Source Code: Many approaches investigate the impacts of changes by reasoning about inheritance relations, method-call behavior, and other dependencies between program entities. Source code files, class packages, classes, methods, statements, and variables are analyzed to predict the propagation of changes. However, such techniques are not applicable in the early phases of software design and requirements analysis, when no source code is available. Their rather technical nature also limits their application to programmers and is of little use for other stakeholders, such as project leaders.

Static code analysis extracts facts from source code to build call graphs, slices, and other representations which are used to assess the impacts of a change. In contrast, dynamic and online approaches instrument the code or compiled binaries to collect information about method executions. These so called execution traces are either analyzed after program execution (dynamic) or on the fly (online) to enable the concurrent assessment of changes.

2) Architectural Models: Williams and Carver [22] highlight the importance of change impact analysis for software architectures to keep their quality and correctness at an acceptable level.

Architectural models, such as UML component diagrams, enable the assessment of architectural changes on a more abstract level than source code. This enables impact analysis in earlier stages of development and in model based development (MBD), which has become more important in recent years. But dependent on the underlying modeling language, even architectural analysis allows for fine-grained results, for example when analyzing detailed UML class diagrams. Typical levels of granularity are systems, sub-systems, components, and classes.

3) Requirements Models: Formalized requirements are the first available artifacts during the software development process, and undergo many changes until the final version of a program has been implemented. If requirements are encoded in formal modeling languages such as UML, GRL [23] or UCM [23], they can be subject of formal analysis as they adhere to a well-structured metamodel. In contrast, if they are expressed as plain text, only textual retrieval methods such as information retrieval (IR) can be applied.

4) Miscellaneous Artifacts: It is an acknowledged fact that changes do not only occur in source code or architectural models. Documentation, configuration files, bug trackers, and similar auxiliary content of software are also subject of frequent changes. However, changes to such entities can also affect the software, e.g. when a configuration file was changed. Performing impact analysis among such entities, mostly different types of files, has therefore become an issue to the research community as well.

5) Combined Scopes: The typical software development process is comprised of different phases, i.e. starting with requirements analysis and architectural reasoning, which are then followed by the implementation. When changing software, impacts do not limit themselves to certain kinds of artifacts. Changing a requirement can affect different architectural models, as well as already implemented source code components. Therefore, comprehensive analysis is required to trace impacts across all available artifacts. This however requires sophisticated concepts that are able to deal with a wide variety of possible types of artifacts.

C. Classification Criteria

Our taxonomy [9] was established after evaluating existing work which is aimed at providing a classification scheme for impact analysis. We reviewed the framework for comparison proposed by Arnold and Bohner [7], a taxonomy for software change established by Mens and Buckley [24], [25], and a taxonomy for impact analysis by Kilpinen [21]. The following listing summarizes our classification criteria, whereas detailed descriptions can be found in [9].

- Scope of analysis - what is analyzed?
- Granularity of analysis - how fine-grained are entities, changes, and proposed results?
- Utilized technique - which algorithms are used?
- Scalability - how scalable is the approach?
- Supported languages - which languages are supported?
- Tool support - is there a tool available?
- Style of analysis - is the analysis performed globally, search-based or exploratory?
- Experimental results - what experimental results are available to judge the performance?

III. IMPACT ANALYSIS: A REVIEW

A. Abbreviations

This rather short section explains the abbreviations, which will be used throughout the review to ensure the usability of its results, especially concerning Table II which contains the results of the classification. All algorithms and techniques which can be found in the following Table I are explained by our taxonomy in [9].

Abbr.	Type	Explanation
PM	Techn.	Probabilistic Models
DG	Techn.	Dependency Graph
MDG	Techn.	Message Dependency Graph
SL	Techn.	Slicing
IR	Techn.	Information Retrieval
TR	Techn.	Traceability
HM	Techn.	History Mining
CG	Techn.	Call Graph
ER	Techn.	Explicit Rules
ET	Techn.	Execution Traces
chg.	Change	unspecified change
vis.	Change	visibility change
typ.	Change	change of data type
val.	Change	change of value
inh.	Change	change of inheritance relation
sig.	Change	change of method signature
mod.	Change	change of modifier
r.	Change	rename
m.	Change	move
+	Change	addition of an entity
-	Change	deletion of an entity
T	Misc.	time complexity
S	Misc.	space complexity

TABLE I
ABBREVIATIONS FOR TECHNOLOGIES, ALGORITHMS AND CHANGES

B. Source Code

This section is focused on approaches which analyze source code. Due to the large amount of studies, we further structure this section according to the techniques used by the approaches, to group approaches according to their similarity.

1) Call Graphs: Changed methods or procedures can affect other source code entities, which either call them directly or indirectly. Analyzing the call-behavior of a system can therefore help to assess the impact of a method/procedure change. Thus, source code is analyzed statically while method calls are extracted and stored in a graph or matrix. This graph then enables developers to estimate the propagation of a given change.

Ryder and Tip [4] developed an approach to check which tests are affected by a change and should be re-executed. In addition, their approach is able to tell which change is responsible for a test's failure, based on call graph analysis. They support fine-grained change operations, such as the deletion of method from a class and implemented their approach in a tool called *Chianti*, which is presented in the work of Ren et al. [26]–[28]. The tool is implemented as Eclipse [1] plug-in but does not seem to be available any longer[2]. *Chianti* performs the following four steps to determine affected test cases. First, it analyzes code edits to obtain sets of interdependent atomic changes. Secondly, it constructs a call graph for each test case which is then used for correlating obtained change sets against the call graph of the original program to determine which test is affected. Finally, it constructs a call graph of the edited version of a program and correlates this graph against the original call graph to determine which changes affect a test. The PhD thesis of Ren [29] continues the work and actually implemented the *Chianti* prototype, and investigates dependencies between atomic changes. Ren implemented a heuristic to rank atomic changes according to their likelihood of affecting a test case failure based on the type of dependency between changes. Ren distinguishes between structural dependencies (sequence when elements are added or deleted), declaration dependencies (java element declarations), mapping dependencies (mapping changes to method-level changes), and syntactic dependencies between atomic changes to exclude changes which do not affect certain tests.

Xia and Srikanth [30] are concerned with measuring internal software artifacts to reflect on the external view of maintainability, and therefore propose a measure to estimate the change impact. Their approach assumes that only source code is available for analysis and that there is no knowledge about change types. They trace changes to program statements by examining the call hierarchy of the code until the change rippled across two levels of the graph, after which further propagation is stopped.

The goal of the work presented by Badri et al. [31] is to increase the precision of call graph based change prediction while keeping the analysis costs on a moderate level. Therefore, they combine static call graphs with static control flow information to so called *Control Flow Graphs*, to overcome the lack of precision of traditional call graph

---

[1] http://eclipse.org/

[2] http://aria.cs.vt.edu/projects/chianti/

based approaches. Their proposed methodology records control flow between method calls to enhance the call graph with information about the call order, and to remove excluded calls in order to remove infeasible paths. *Control call graphs* (CCG) are constructed by analyzing source code while ignoring all statements and instructions which do not invoke a method call. CCGs are further enhanced with sequence information to enable the generation of compacted sequences, which are then used to further prune infeasible paths. As a result of this process, the behavioral profile of a program is obtained. The proposed approach is called *CCGImpact* and implemented in the *PCIA* tool. However, their approach does not yet consider object oriented specifities [31], such as inheritance.

2) Dependency Analysis: There exist a variety of dependencies between source code entities, such as control, data or inheritance dependencies. They can be extracted by static source code analysis and either stored in a graph or a matrix. Based on dependencies between software artifacts, one can estimate the change propagation between them.

Briand et al. [32] investigate the complex nature of object oriented programs. The OO paradigm results in large numbers of dependencies when analyzed with traditional dependency analysis, which in turn leads to an explosion of possible impacts computed by dependency analysis. Therefore, they investigate whether dependency-based coupling measures can be used for impact analysis. *Coupling between object classes* (CBO) and *data abstraction coupling* (DAC) are two examples for such a coupling measures. Two classes, A and B, are coupled via CBO if B calls a method of A. They are coupled via DAC if B contains an instance of A. Their approach computes all possible coupling measures and ranks classes according to achieved values, where a high ranking indicates a strong coupling and therefore implies that an entity is impact by a change to the coupled entity.

The work of Kung et al. [33] is concerned with code changes in object oriented libraries. They developed a model to capture and reason about changes, as the understanding of combined relations and dependencies between classes is more complicated than ripple effect analysis in procedural languages. Their proposed model describes control structures, inheritance relations, and the like between classes, methods, and attributes. The model is further build upon three types of diagrams, namely *Object Relation Diagrams* (ORD), *Block Branch Diagrams* (BBD), and *Object State Diagrams* (OSD). These diagrams are generated from source code and applied for analysis, using the class firewall concept. Attributes which are affected by a change are detected by computing the differences between the source code and the current BBD, whereas impacted methods are indicated by removed nodes and edges in the BDD. Impacted library classes are determined by ORD analysis.

Rajlich [34] discusses two approaches of change propagation, which are based on program dependencies and have been implemented in the tool *Ripples 2*. The utilized dependency graph is computed by a code parser which scans project files and identifies dependencies. Changes in the dependency graph are modeled by graph rewriting, whereas the evolution of the dependency graph is modeled as a sequence of snapshots. The proposed ”change-and-fix-approach” consists of two steps, which are repeated until the change has been implemented. First, the developer changes an entity which then triggers a dependency analysis to propose possible impacted entities. This process is repeated, as long as the developer is changing further entities. In contrast, the ”top-down-approach” marks entities as either *top elements* if they do not support other entities or *bottom elements* otherwise. The analysis then starts by first examining *top elements* and then moving along the dependencies if no change is required in the top-elements.

The approach of Pirklbauer et al. [35] is based on the tool developed by Fasching [36] for visualizing software dependencies, and is comprised of six steps as proposed by Bohner [2], [37]. Impacted elements are computed by analyzing the dependency graph of a program while using three parameters to prune infeasible paths: dependency type, minimal importance of dependency, and maximum dependency depth. However, the developer has to inspect the remaining graph manually and decide which entity has been correctly identified as impacted. In the underlying master's thesis of Fasching [36], the author highlights the importance of visualizing dependencies to enable developers to understand the changes. The thesis outlines important requirements for impact and dependency visualization, such as interaction, navigation, zoom, merge, hiding, reload of data, and focus, and it also discusses various visualization techniques and compares existing tools for impact visualization.

Similar to the work of Kung et al. [33], Zalewski and Schupp [38] are also concerned with assessing the impacts of library changes. They propose a methodology called *conceptual change impact analysis* (CCIA) which is based on the principle of pipes and filters. In a first step, they locate changes which impact conceptual specifications and then apply optional filters to refine the output to detect specific kinds of impacts. Therefore, they provide two filter algorithms: one to detect the impact of a change to the degree of generality, and one to detect the compatibility to different versions. The second step involves the implementation of the change and the identification of differences between the original and altered version of the program. Next, a dependency graph is constructed and nodes are annotated with information gained by the differencing process (e.g. they are marked as ”added”). The last step involves a depth-first-search to propagate the impacts of a change through the graph, which uses the additional node information to prune the propagation of changes, e.g. when reaching a node which is marked as ”deleted”. As a further contribution, they provide two algorithms to search for specific kinds of impacts, namely *constraints change* which checks whether requirements for algorithm parameters changed, and *concept compatibility* which checks whether a concept is compatible between versions.

Petrenko and Rajlich [39] are concerned with enhancing impact analysis to cope with variable granularity of analyzed software artifacts. Therefore, they construct a dependency graph of the program and use marks to annotate nodes as either *changed*, *propagates*, *next*, *inspected* or *blank*. If an impacted element is found, it is the programmer who decides whether the impact propagates at the same, a coarser or even a finer level of granularity. All children of the impacted element will be marked, if the programmer refines the granularity, e.g. from "class" to "method". In contrast, all parent elements will be marked if the programmer coarsens the granularity, e.g. from "attribute" to "method". If no granularity is sufficient, the programmer can select entire code fragments where all encompassing entities will be marked and used for further impact propagation. They extended the *JRipples* tool to cope with this variable approach.

Black [40] is concerned with reformulating the ripple-effect algorithm as proposed by Yau et al. [41], and implemented the improved version in the *REST* tool to analyze C programs. *REST* builds a dependency matrix of the program and applies four different McCabe complexity measures to compute the ripple effect and stability measures for the source code. Black performed an evaluation to gauge the correlation between the original algorithm and the *REST* tool and concluded that both match, whereas *REST* achieves more accurate results.

The aim of Bilal and Black [42] is to improve the ripple effect algorithm introduced by Yau et al. [41] in 1978, by optimizing the ripple effect metric. Their analysis is based on two types of change propagation to calculate the ripple effects, namely intra-module change propagation and inter-module change propagation. Both types of change propagation are implemented in four different algorithms. The first algorithm concatenates all the code within a class apart from local method calls, and computes the ripple effect between this class and others. The second algorithm computes ripple effects across methods and classes, but ignores change propagation within methods. The third computes ripple effects between methods, within methods, and between classes. The forth and last algorithm computes ripple effects across the methods of each class, by ignoring the change propagation between classes. However, the authors did not provide any details on the results achieved with each algorithm and based on the paper one cannot judge which is more suitable.

Lee et al. [6] utilize three different object oriented dependency graphs of which they compute the transitive closure to identify impacted elements. The authors propose the concepts of *intra-method data dependency graphs* to calculate impacts on entities inside method bodies, *inter-method data dependency graphs* to calculate change dependencies between methods, and *object oriented system dependency graphs* which are used to calculate the change impact at system level. Furthermore, they distinguish between four different types of impacts between two related entities, namely *contaminated* (both are impacted), *clean* (both are not impacted), *semi-contaminated* (the source does not impact the target, but the source can be impacted), and *semi-clean* (the source is not impacted, but it propagates changes to the target). These types are used to assign weights to relationships among entities, according to the type of impact relations between them. The total change impact weight is then computed as the sum of all weights which are assigned to the relations between two entities. Finally, the total change impact weight is assigned to all three graphs to enable the computation of impacts.

The question which types of changes can occur in object oriented systems and how they affect other classes is the driving motivation for the work of Li and Offut [43]. Their proposed algorithm computes the transitive closure of the program dependency graph, before applying the actual analysis procedure. The analysis starts by grouping all classes that are related to a changed class into a set of affected classes. Secondly, the union of all sets is computed. The computation of the total effects is then comprised of the following three sub-computations: the identification of effects within a class, the identification of effects among related classes (clients), and finally the identification of effects caused by inheritance relations between classes.

Beszédes et al. [44] propose the use of *Static Execution After* relations (SEA) to approximate static slicing with less costs. The computation of SEA relations is achieved by analyzing methods to unveil hidden dependencies between classes. This is accomplished by constructing a *Component Control Flow Graph* (CCFG) for each method, which collects method calls by static source code analysis. The CCFG is build by adding one entry-node for each method and one component-node for every other method that is called within the method itself. Both nodes are then connected with "control flow" connections. The algorithm then computes all SEA relations by determining the transitive calls for each method. Then, all identified methods are processed again to compute the sequence information by ordering components topologically, and determining the set of methods which are called in each component.

Jász et al. extend [45] the work of Beszédes et al. [44] by introducing *Static Execution Before/After* relations (SEA/SEB), where SEB are the inverse to SEA. The SEA and SEB relations are computed based on the *interprocedural component control flow graph* (ICCFG), which is comprised of call site nodes and control flow edges that exist between them. An experiment has shown that the concept of SEA/SEB results in decreased computation time, but also in a loss of precision of about 4%. One of the programs analyzed in the scope of this experiment (Mozilla) could not even be analyzed with traditional dependency analysis due to space and time requirements, but with SEA/SEB relations.

Supporting development activities with solid tool support is the main goal of the work presented by Chen and Rajlich [46]. They introduce their tool *RIPPLES*, which uses an *Abstract System Dependency Graph* (ASDG) to estimate the propagation of changes. *RIPPLES* combines dependencies obtained through static analysis with conceptual dependencies, as not all dependencies between code-components can be captured by static analysis. Nodes of the ASDG are comprised of program variables, methods, and data types, whereas edges in the graph are based on control flow information. The size and complexity of the ASDG is reduced by removing statements containing no function call or variable modification. Next, nodes of the ASDG are either marked as *unmarked* (default), *candidate* or *changed*, where edges are also marked as either *unmarked* (default), *forward* or *backward*. When a programmer inspects the ASDG and updates program entities, assigned marks are updated according to the actions taken by the programmer to signal possible impacts, e.g. when deleting an entity.

Gwizdala et al. [47] propose the *JTracker* tool to guide programmers while changing software. *JTracker* analyzes the program and builds a dependency database, which is updated after each operation. The tool considers inheritance, aggregation, and other relations between classes as dependencies. *JTracker* marks classes which require inspection by the programmer until no marked class is left in the program database. Gwizdala et al. analyzed a program consisting of 400 classes in a case study, but unfortunately they did not provide any results in terms of precision and recall.

The traditional tradeoff between achieved precision and required computation time is the motivation for Bishop [48] to propose an incremental approach which requires less analysis effort to compute impacts. The main goal is to reuse earlier results of impact calculations, and to update them according to changes made to the program. A dependency graph of the program is constructed based on a method input-output-mapping and enhanced with static data flow information which are obtained from method bodies. The incremental approach stores the results of each analysis process to enable an incremental search for impacted entities, which results in reduced costs.

3) Program Slicing: Slicing is one application of static source code analysis and is build upon code dependency analysis. A comprehensive study on slicing can be found in the work of Tip who conducted a survey on the topic [49]. Slicing removes all program statements which are not related to the slicing criterion, i.e. which do not affect the state of a variable and thereby being of no use for impact analysis. A short tabular comparison of several slicing tools which can be used for impact analysis is contained in the work of Korpi and Koskinen [50] which will also be discussed in this section.

The work of Gallagher and Lyle [51] is concerned with forming slice-based decompositions of programs, containing only those parts which are affected by a change. They use the concept of decomposition to break the program into manageable pieces and to guarantee that there are no ripple effects. Decomposition slices are constructed by applying an algorithm which was originally designed for dead code detection and locates all instructions which are useful regarding the slicing criterion. The detection algorithm marks all output instructions as *critical* and traces instructions that impact the output statements (called "use-definition"). Detected relations are then examined to build the actual decomposition slices, by forming a lattice of slices for each variable and ordering them by set inclusions. Finally, all output statements are removed from the slices, as they do not contribute anything to the variables. The result of this process is a slice which only contains impacted code elements.

Visualizing impacts is a useful technique for developers to understand which entities are affected by a change. However, current visualization techniques which are based on acyclic graphs flood developers with too much information, and are less useful for slicing. The aim of Hutchins and Gallagher [52] is therefore to reduce the amount of slicing information which must be displayed to assess the impact of a change. They introduce the concept of inference between variables, which distinguishes between strong (directly related) and weak dependencies (mutually dependent on other variables), and result in two different impact graphs. They further propose a set of interface features to reduce the size of slices. Their approach allows to hide any variable of a particular procedure, hide variables with empty slices ($|slice| \leq 2$), group equivalent variables of the same procedure into one entity, group all slices of all variables of a procedure into one entity, and hide variables without relationships to current selection.

Tonella [53] investigates how decomposition slice graphs can be combined with concept lattices which group elements sharing common attributes, to increase the precision and recall of impact analysis. Tonella combines nodes from both approaches via *and* and *or* relations to obtain a lattice of decomposition slices (and-lattice, or-lattice). Two decomposition slices are considered to interfere, if there exists a non-empty intersection of them, i.e. decomposition slices provide strong-intersections, whereas lattices provide weak and strong-intersections. The identification of nodes which are directly affected by a change is achieved by traversing the lattices upwards, as all upwards nodes share the same line of code.

The concepts of dependence clusters and dependence pollution are introduced in the work of Binkley and Harman [54], who also propose a visualization technique for locating such clusters. Their approach is based on the size of computed slices, i.e. they create slices for each variable and check whether the slices have the same size. If so, such variables contribute to a cluster and changes to one variable in the cluster are likely to cause changes to other variables of the same cluster. They implemented their idea in the concept of *monotone slice-size graphs* (MSG), which is used to visualize detected clusters. An experiment revealed that 80% of all studied programs contained clusters which consist of 10% of the original program.

Korpi and Koskinen [50] developed an automated PDG-based forward slicer for Visual Basic called *GRACE*, to enable impact analysis for Visual Basic programs. *GRACE* applies static forward slicing to capture all possibly affected parts of the program, and consists of two main components. The parser translates Visual Basic programs into *abstract syntax trees* (AST) and symbol tables. The PDG-generator finally converts the ASTs into graphs. *GRACE* combines single graphs to system graphs, and performs a straightforward reachability analysis on the PDG to compute impacted code entities. An evaluation based on a program consisting of 18700 LOC revealed that parsing the code and generating the PDG took five seconds with *GRACE*.

Preprocessor macros are a useful concept in programming languages. However, they result in sometimes incomprehensible and unmaintainable code. Also, there is no impact analysis approach available which is able to analyze the effects of macro changes. Therefore, Vidács et al. [55] propose a slicing algorithm for C++ preprocessor macros to enable impact analysis on macro level. They build a macro dependency graph which is used to compute dynamic forward slices, as they contain possibly affected program entities. The edges of this graph are based on macro dependencies, whereas nodes are based on macro invocations and macro definitions. Their approach is enhanced with the concepts of coloring and multiple edges to provide support for later defined and re-defined macros in dynamic forward slices.

Santelices and Harrold [56] are concerned with the question how static slicing or slicing in general can be enhanced to reduce the set of proposed impacts by utilizing probabilistic algorithms. Their research is based on the following three observations:

1) Not all statements have a similar probability of being impacted by a certain change.
2) Some data dependencies are less likely to be covered.
3) Data dependencies propagate changes with a higher probability than control dependencies.

The probability that A is affected by B is computed by statically analyzing whether the execution of a program reaches A after executing B, if there are sequences of data or control dependencies between B and A, and if a modification of a program state propagates from B to A. Standard slicing procedures are used to generate a dependency graph that covers all possible executions, which is then used to create an interprocedural dependency graph. This graph is annotated with the probabilities that dependencies are affected by a change and will propagate the change themselves. After this step, each path is annotated with the probability of propagating changes from the source node to the target node, which is composed of all single probabilities on the path. This composition is achieved by computing the coverage and propagation probabilities and executing a data-flow analysis on the PDG to combine all dependencies between the source node and the target node. The decision whether an entity is impacted or not is then dependent on computed probabilities.

4) Execution Traces: In contrast to static call graphs, dynamic execution traces contain only those methods which have been called during the execution of a program. Similar to call graphs, they allow to assess the impact of a method-change by analyzing which methods were called after the changed method, thus being possibly impacted too. Approaches which rely on dynamic execution data were established to overcome the limitations of static slicing (expensive) and call graphs (imprecise).

Law and Rothermel [57], [58] developed two dynamic impact analysis algorithms: *PathImpact* and its improved version *EvolveImpact* which are explained in this section.

PathImpact [57] instruments methods to collect method entry and method exit events as execution traces. However, traces suffer from duplicated entries and have to be compressed, since methods can be called more than once. Law and Rothermel compressed traces to DAGs using the *SEQUITUR* compression algorithm to overcome this problem. A traversal of the DAG reveals all methods which were called after a changed method was called, or in which a changed method returned into. All these methods are therefore considered as being potentially impacted.

EvolveImpact [58] was proposed to overcome the problem of updating the DAG after changes, as rebuilding the entire DAG for larger programs is an expensive task. The goal is to incrementally update the DAG by observing changes to test suites and system components. Therefore, each test case is tagged with a unique identifier which also appears at the beginning of each trace and is supplemented by special ending symbols. These keys and ending symbols are then used to remove traces from the DAG if they contain changed methods and indicate that some traces should be refreshed. Law and Rothermel developed a modified version of the *SEQUITUR* compression algorithm called *ModSEQUITUR* to cope with these new symbols and keys.

Orso et al. [59] claim that program testing and analysis under development conditions is not realistic, since real world factors such as different hardware, different users etc. have a big influence on the behavior of software. They are concerned with gathering ”field data” from real users under real world conditions, to evaluate software and perform impact analysis. They investigate the use of the Gamma-approach for impact analysis, which has been introduced by the same authors. This approach enables remote analysis and the collection of field data from real users. Developers have to instrument their programs to collect dynamic data (i.e. execution data) in a lightweight manner. The users execute the programs and send back the execution data (coverage data at block and method levels). The developer then computes dynamic slices, which are based on execution data of entities that traversed a changed entity. This kind of analysis also allows to estimate the impact of a change on different users by asking questions like ”How many % of the users will be impacted by this change?”. Orso et al. compared their approach against static slicing and call graphs and concluded that real field data is different to synthetically computed data.

Breech et al. [60] proposed an approach for online impact analysis. The authors instrumented a compiler to add callbacks to each function, in order to be able to gather execution data during runtime. Once the data has been collected, they analyze the call stack to judge whether a function is affected by a change of other functions. All functions which were called by a changed function are considered as impacted, and all functions which remain in the call stack after the changed function’s return event are also considered as being impacted.

In further work, Breech et al. [61] are concerned with combining the precision of static techniques and the analysis speed of dynamic techniques. As a first step, they build an influence graph of the program, as changes can only ripple through return values, parameters, and global variables. The graph is comprised of methods (nodes) and their relations (edges), and is used to compute the transitive closure of the source code. A dynamic analysis is then performed and results are merged with information inferred from the influence graph. That is, only those methods are considered as impacted, which are connected in the influence graph with the same type of relation as inferred by the dynamic analysis. However, case studies have shown that this approach can only achieve a 3-5% gain of precision compared to *PathImpact* [57].

Apiwattanapong et al. [62] are concerned with overcoming the performance limitations of *PathImpact* [57] and the lack of precision of *CoverageImpact* [59]. Therefore, the authors introduce the concept of *Execute-After sequences* (EA). They achieve their goal by simplifying recorded traces in terms that they only record the first and last event of each method, i.e. they neglect *method return* events and instead only record *method return into* events. Each EA-sequence is enhanced with additional timestamps to determine impacted methods. If a method is changed, all possibly impacted methods are determined by searching for methods whose timestamp is equal or higher than the first timestamp of the changed method. The algorithm proposed by Apiwattanapong et al. is called *CollectEA*, and results of several case studies have shown that *CollectEA* is almost as precise as *PathImpact*, but only slightly more expensive than *CoverageImpact*.

Existing dynamic techniques introduce too much overhead in terms of space and time costs, as stated by Huang and Song [63]. Therefore, the authors propose a dynamic impact analysis technique which collects *method return into* events, rather than *method return* events. The aim of the authors is to achieve the same precision and recall as *PathImpact* [57] and *CollectEA* [62], but with less time and space complexity. Their algorithm retrieves information about the execution order of methods by correlating method events with methods executed after the event. These information are then used to reduce the redundancy in execution traces. Execution events which do not traverse a changed method are skipped, and methods whose events are listed before any event of a changed method took place are removed as well. As a result of this process, only methods executed after the execution of a changed method remain in the estimated impact set.

Beszédes et al. [64] introduce a new coupling measure, *Dynamic Function Coupling* (DFC), in order to take both directions of an *Execute-After* relation into account. The basic idea of DFC is to utilize the closeness of two functions to estimate whether one impacts the other. The closeness of two functions is determined for each pair of functions, by computing the distance between their call levels, therefore using forward and backward *Execute-After* relations. The DFC measure is used to select function pairs, whose distance is below a certain cut-off level (threshold). The DFC values are computed based on a dynamic call tree, which is comprised of method entry and method exit events. Each function pair selected according to the DFC measure is then considered as being impacted by a change to one of the functions.

Gupta et al. [65] propose a new dynamic algorithm to compute the impact of a method change, which allows to trace the impact of a change on other program variables: *CallImpact*. Tracing impacts relies on dependencies between program entities, which can be distinguished based on their type and usage, based on the types of paths between definition and usage, and based on the reach of the data dependency. They build a *control flow graph* (CFG) of the program while inserting variable definitions or assignments as nodes. This control flow graph and the different types of dependencies between the nodes enable them to distinguish between directly and indirectly impacted entities. Direct impacts are computed by collecting all usage traces of the changed node. Indirect impacts are computed by analyzing previously collected directs sets and further tracing them, according to the type of dependency.

Gupta et al. [66] extended their work, and additionally propose a classification of changes to improve impact detection. They distinguish between functional changes (changed statements which affect functions), logical changes (control-flow changes), structural changes (addition/deletion of code entities) and behavioral changes (change of execution order, change of program entry and exit). Their revised algorithm works as follows. The original and the modified version of the program are analyzed for differences, which are stored together with their respective CFGs in a database. A classification algorithm is applied on the collected differences to calculate functional impacts based on the connectivity of changed statements in the CFG. This step adds all changed statements and their direct impacts to the impact set. As a second step, all logical changes are computed by analyzing statements in the CFG which resulted in control flow changes. All these statements and those which depend on them via definition or usage are also added to the impact set. Structural changes are then computed by identifying all added or removed statements of the original code, whereas behavioral changes are computed by adding all statements to the impact set which cause behavioral changes in the original program.

Hidden dependencies play an important role in software evolution and even exist in well structured software. Vanciu and Rajlich [67] propose an approach for dynamic data flow analysis to uncover such hidden dependencies for impact analysis. Their approach is based on *Executed Completely After* (ECA) relations, which state that methods calling others must be terminated, before the other method is executed. They also introduce the *Interclass Executed Completely After* relations (IECA), which extend ECA relations by stating that both methods must not belong to the same class. ECA/IECA relations consist of preconditions (required state of memory before executing a method) and postconditions (state of memory after executing a method). The potential hidden dependencies between two methods *f* and *g* are revealed, if there is one post-condition of *f* which is implied by at least one pre-condition of *g*, which they call *significant precondition conjunction* SPC(f,g). Finally, a potential hidden dependency is a true hidden dependency, if both methods share the same domain or programming concept.

5) Explicit Rules: Design, domain, and expert knowledge can be used to form strict impact rules, which determine which entities have to change if a certain entity changes. For example, if an interface is changed, all classes which implement this interface must be changed as well, dependent on the type of change (e.g. deletion of a method).

The aim of the work of Han [68] is to support developers with a framework for software change management, which is integrated into the software engineering environment to enable impact analysis for forward engineering and reengineering. The proposed impact analysis approach combines automated techniques with guided user intervention, by augmenting the kernel of the software engineering environment with change management features. The impact analysis process is accomplished as follows. First, the approach extracts a dependency graph from the program and then applies a set of propagation rules to determine impacted entities. The obtained set of possibly impacted entities is then presented to the developer, who is then able to decide which one to implement.

Chaumun et al. [69] are interested in measuring the influence of high-level design on maintainability, and how interclass dependencies influence changeability. They propose a change impact model which is defined at the conceptual level, and map it to the C++ programming language. The conceptual model is comprised of *components* (classes, methods, variables), *relationships* between them (association, aggregation, inheritance, invocation), and 66 *change types*, such as the addition of a component. The impact of a change now depends on two main factors in their model: the type of change and the type of relation between both code entities. Chaumun et al. define a set of impact rules which are comprised of change types and relation types, and are expressed as boolean equations.

Many coupling measures do not capture all aspects of object oriented software, such as multiple inheritance. Therefore, Arisholm et al. [70] propose a new dynamic coupling measure which can be used for impact analysis in object oriented software. Dynamic coupling measures are collected in two different ways. First, the program is executed and message traces and other information supplied by the Java Virtual Machine (JVM) are used, rather than instrumenting the source code. Secondly, dynamic UML diagrams (e.g. sequence diagrams) are analyzed to extract information about the dynamic behavior of the system. Based on the monitored behavior, a set of 12 coupling measures built on rules is derived.

The main motivation for the work of Sun et al. [71] is that different change types result in different impacts, and therefore require specific rules to compute the impacts. They propose a static impact analysis approach, which considers different change types and impact mechanisms, by classifying change types at class and class member level. Their approach starts by constructing an intermediate representation of Java programs, called *Object Oriented Class and Member Dependence Graph* (OOCMDG), which contains classes, methods, variables, and their dependencies. The proposed model for change propagation considers different change types and dependency types for classes, methods, and variables which are used to formulate impact rules. The starting impact set (SIS) [9] is obtained by performing forward and backward walks on the OOCMDG, starting at the changed entity. The estimated impact set (EIS) [9] is obtained by classifying changes according to their change types and then computing the union of all change types for each entity. Based on this union of change types, the appropriate impact rules are chosen which then compute the final impact set for a given change.

6) Information Retrieval: Information retrieval (IR) techniques exploit natural languages by searching for similar terms in different documents, to infer a relation between them. The interested reader might be referred to a recent survey of Binkley and Lawrie [72] on the application of IR in software evolution and maintenance.

In contrast to most impact analysis approaches that estimate which software entities are affected by a change, Antoniol et al. [73] propose an approach to identify entities which must initially be changed to fulfill a change request, i.e. the identification of the starting impact set (SIS) [9]. They apply two IR techniques (a vector-space model and a probabilistic model) to score documents associated with the software, according to their likelihood of belonging to the SIS of a change request. First, their approach extracts ”meaningful” words from documents, which are then matched with code entities. Thereby, they are ranking the documents to limit the amount of retrieved documents to the relevant ones only. The whole process consists of two steps, and starts with identifying the set of high-level documents and then using documentation-to-code traceability links to identify source code components which are initially affected. They performed an early evaluation while studying the LEDA C++ library and achieved a precision of 48% with 70% of recall.

Uncovering evolutionary patterns from source code can assist with impact analysis, if developers are aware of frequent change patterns and involved code entities. The work of Vaucher et al. [74] is therefore aimed at identifying such patterns on the level of classes, using IR techniques. They calculate the *level of change* for each class evolution by retrieving information regarding the implementational and functional changes from code, which are then stored in a vector. Their technique retrieves the relative implementation changes by counting the number of added, removed, and modified instructions of methods. Functional changes in public interfaces of classes are identified by dividing the number of added and removed public methods through the total number of public methods. The total change of a class is then computed as the sum of all relative and functional changes. Once all information have been gathered, classes are clustered according to their change behavior, where *dynamic time warping* (DTW) is applied to identify similar groups of class-clusters (patterns). Class patterns are further grouped according to categories such as *usual suspects* (classes which frequently change together), *code stabilizations* (new classes that require a few versions before becoming stable), *punctual changes* (classes grouped due to change in specific version), and *common concern* (classes implementing same concern).

Similar to Arisholm et al. [70], Poshyvanyk et al. [75] are also concerned with overcoming the limitations of static coupling measures, and claim that different dimensions must be covered by a reliable coupling measure to be of use for impact analysis. They propose a new set of coupling measures for classes, which are based on IR techniques. Their approach utilizes overlappings in identifiers, names, and comments to detect similarities between code entities using *latent semantic indexing* (LSI). However, they do not use common IR techniques such as word stemming, expansion of abbreviations, and word replacement to limit the amount of extracted terms. LSI creates a term-by-document matrix which captures the distribution of words in methods, and which represents each document as a vector in the LSI subspace. Based on this matrix, they compute the cosine between vectors to measure the conceptual coupling between methods and then apply coupling measures to rank classes for impact analysis, based on their coupling value.

7) Probabilistic Models: Probabilistic models, such as Markow chains and Bayesian Belief Networks (BBN), allow to model the propagation of changes based on well explored mathematical models and theorems. Thus, they allow to compute the probability of an entity being impacted by a change.

Zhou et al. [76] state that most MSR approaches neglect code ownership and the age of changes in their analysis process, as code of the same author is more likely to change as well and entities affected by recent changes are candidates which most likely change again. They propose an approach which is based on BNNs, and is able to predict change couplings. Changes are mapped onto different significance levels, which are based on several complexity indicators (e.g. nesting depth) or whether they modify or preserve functionality. First, the change history and transactions are recovered from version control repositories in order to extract fine-grained code changes, which are then classified. During this data import, a sliding window approach is used to extract co-changes for preparing a static source code dependency model. The second step consists of feature extraction and instance generation, by selecting features that influence whether entities might co-change. The number of features per entity is counted to classify them into impact groups of *low*, *middle*, *high*, and *uncertain*, by taking into account the change significance level, co-change frequency, age of change (old = stable, latest = most likely to change), and the author. The third phase is comprised of training and validation, therefore using the selected entities and their instances to build the structure of the BNN. This is achieved by applying the *K2* algorithm and the *SimpleEstimator* algorithm for calculating feature frequencies, which are used to derive the probabilities.

To evaluate how each class will be affected by a change, Tsantalis et al. [77] propose an approach to measure the change proness of object oriented software. They distinguish between three ways how a class can be affected externally by inheritance, references or dependencies. Furthermore, they also consider the "inner axis", i.e. changes within the class itself. Their proposed tool computes the probability of a change for each axis while scanning the development database of the software, where the developer has to classify changes as either *internal* or *ripple effect* (external). The calculation of probabilities starts with classes which are not part of a dependency cycle to enable the stepwise computation of classes, where either their internal or external axis belongs to a dependency cycle. In the case of two classes being mutually dependent, the following additional steps are performed. First, all relations between the classes are temporarily removed to calculate probabilities for single classes. Finally, the relations are added again to adjust the probabilities accordingly.

Abdi et al. [78], [79] use static analysis to uncover potentially affected classes and apply machine learning to score each detected, possible impact with probabilities. They consider 13 change types for classes, methods, variables, and distinguish between four types of links between entities (association, aggregation, inheritance, invocation). The impact depends on the type of link between two entities and the type of change applied upon them. It is computed by static code analysis based on couplings between classes, using several coupling metrics. Four different machine learning techniques (J48, Jrib, PART, NBTree) are then used to compute the likelihoods of previously discovered potential impacts. Their approach is implemented in *PTIDEJ* for Java, and achieved a precision of 69% in a case study of a system consisting of 394 classes.

Further work of Abdi et al. [80], [81] is concerned with the problem that causality relations between software artifacts are still not explained. Therefore, they propose a probabilistic approach using BBNs to model impact propagation on the level of source code. The proposed methodology consists of three steps. First, the structure of the BNN is constructed by analyzing the structure of the software and by manually assigning initial probabilities. Secondly, the parameter affectation is computed by distinguishing between *entry variables* which can be directly measured, and *intermediate variables* which are influenced by parent nodes and are obtained through machine learning. The third and last step performs the actual bayesian inference and updates all conditional probabilities assigned to the nodes of the BNN.

Mirarab et al. [82] also propose the usage of BBNs for change impact analysis. Their proposed solution consists of the three steps *data extraction*, *network generation*, and *network analysis*. *Data extraction* recovers changes and extracts dependencies from repositories through static coupling measures. Meanwhile, IR heuristics are used to limit the number of dependencies, and co-change patterns are extracted from version control repositories. Secondly, *network generation* uses three different models for prediction, the *bayesian dependency model* (BDM), *bayesian history model* (BHM), and the combination of both, the *bayesian dependency and history model* (BDHM). The creation of the models is based on the previously extracted information and uses parameter learning to incorporate historical information into the models. Last, the *network analysis* is executed by applying bayesian inference to predict changes from the network models, while using the *EPIS* sampling algorithm. A case study conducted on a 263 kLOC system revealed a precision of 63% while achieving a recall of 26%.

Similar to the work presented by Poshyvanyk et al. [75], Gethers and Poshyvanyk [83] are concerned with detecting conceptual couplings which are encoded in identifiers, comments, and other attributes. They propose a new probability-based coupling detection technique, *Relation Topic based Coupling* (RTC), to compute the coupling between classes. The applied RTM model is an unsupervised probabilistic topic modeling technique, which associates topics with documents by predicting links between documents, based on topics and relations among documents. RTM extends the *latent dirichlet allocation* (LDA) technique and represents the software as a collection of documents, where a word is the basic element, e.g. an identifier. In the RTC model, each document corresponds to a class and is therefore a collection of words (identifiers, methods etc.). They conducted a case study on 13 programs, where Eclipse with 1.9 mLOC was the largest among them and achieved an average precision of 12% and an average recall of 45%.

8) History Mining: History Mining is one possible technique for impact analysis and related to MSR [19]. It mines clusters or patterns of entities from software repositories which often changed together, as a change to one entity of a cluster is likely to affect all the other entities within this cluster as well.

Fluri et al. [84] acknowledge that MSR is able to detect patterns of co-changing entities. However, most approaches are not able to tell which co-changes were caused by structural changes and therefore being important for impact analysis, and which ones were caused by other, less important or even non-relevant changes. They present an approach which adds structural information to change couplings to improve history-based impact analysis. Their approach consists of the following four steps. First, modification records are retrieved from CVS to build a *release history database* (RHDB). Secondly, change coupling clusters are calculated based on the information stored in the RHDB. Thirdly, subsequent CVS versions are structurally compared, using the following two techniques: the comparison of their ASTs and the comparison of code entities (classes, methods, etc.). Finally, change coupling clusters and structural differences are used to filter changes that were not caused by structural modifications. An early experiment with a 26 kLOC Eclipse plug-in has shown that the distinction between structural changes and other changes can help to reduce the amount of mined change couplings, thereby increasing the precision of impact analysis.

Fluri and Gall [85] continued their work presented in [84] and analyze the impact of different change types. They propose a taxonomy for change types and define source code changes as tree edit operations on the AST of a program. They further classify changes according to their significance level, which can either be *low*, *medium*, *high* or *crucial*. For example, they consider local changes to have a *low* significance, whereas interface changes are considered to be *crucial*. Their approach assigns a label to each source code entity in the AST, which represents the type of the entity. They also add a textual description (called *value*), which contains the actual source code, e.g. a method call with parameters. As a next step, they extract change couplings from the RHDB which have a similar total significance level, and weight them according to the number of transactions they occurred in. The remaining change couplings are then the most likely candidates for being impacted by changes to coupled entities. The approach has been implemented in the *ChangeDistiller* plug-in for Eclipse, and was evaluated in a case study containing 1400 classes.

In earlier work, Gall et al. [86] proposed an approach to extract software evolution patterns and dependencies from CVS data. Their proposed methodology, *QCR*, investigates the historical development of classes by analyzing the structural information of programs, modules, and subsystems in combination with version numbers and change information obtained from CVS. *QCR* consists of three analysis methods: the *quantitative analysis* (QA) to analyze the change and growth rate of modules, the *change sequence analysis* (CAS) to identify common change histories of modules, and the *relation analysis* (RA) to compare modules and identify dependencies. The RA methodology is able to detect couplings which refer to similar change patterns among different parts of the software, to identify classes which were most frequently changed together. The detection is based on the comparison of class changes, regarding the author, date, and time of the change, and uses a time window of four minutes. The underlying assumption of this analysis is, that all changes implemented by the same developer at the same day point towards a logical coupling between involved entities. The more congruities are found, the stronger is the observed coupling. For example, if the average class was changed five times a day, the analysis is focused on classes which were changed more than five times a day.

Zimmermann et al. [87] apply data mining techniques to uncover couplings between fine-grained entities to guide programmers among related changes. Their approach, which has been implemented in the *ROSE* tool, consists of two steps. First, the preprocessing extracts data sets from CVS and maps changes to concrete entities of the program. The second step involves the actual mining process to uncover association rules between related program entities. Each analyzed entity consists of three attributes, namely *category* (e.g. file, method), *identifier*, and *parent entity* (can be NULL). The process of mining association rules is then based on these entities and different change types (e.g. added, removed, altered), and uses the *Apriori*-algorithm to compute rules from the data sets. Finally, mined association rules are interpreted based on two different measures: *support count* (number of transactions this rule has been derived from) and *confidence* (how often did the rule predict changes correctly).

Treating history as an explicit first class entity requires the existence of a (meta-)model for software evolution, which is proposed in the work of Gîrba et al. [88]–[91]. The authors define four characteristics for categorizing class evolution [89], [90], namely *age of hierarchy*, *stability of inheritance relations*, *stability of class size*, and *balance of development effort* to define history as a sequence of versions of the same kind of entities. Based on their *Hismo* metamodel, they developed an approach to characterize and visualize the evolution of classes [89], [90] called *Class Hierarchy History Complexity View*, which utilizes tree layouts and polymetric views, and two impact analysis approaches: *Yesterday's Weather* [92] and the work presented in [93].

Yesterday's Weather [92] is build on the everyday phenomena that today's weather can be used to forecast tomorrow's weather. This also applies to software evolution, where previous changes can be used to predict future changes. The approach is build on top of two assumptions: not every change is useful, and classes which changed in the most recent past are most likely to change again. Class changes in general are measured by monitoring changes to their total number of methods. The authors measure the *evolution of the total number of methods* (ENOM), the *latest evolution of number of methods* (LENOM) which is similar to ENOM but ranks changes according to their time stamp, and the *earliest evolution of number of methods* (EENOM) which is the opposite to LENOM. The detection of classes as potential candidate is then based on intersections between EENOM (classes that did change) and LENOM (classes likely to change). The final value of *Yester Weather*'s forecast is then computed by counting the hits for all versions and divide them by the total number of possible hits.

Gîrba et al. [93] use *formal concept analysis* (FCA) to detect common change patterns, which affect many entities at the same time. They utilize a historical measure to detect changes between two versions by identifying an excerpt of history in which a certain change condition is met, built upon the Hismo model. For identifying changes, FCA uses a matrix to associate entities with properties. The property matrix of two versions is then compared using logical expressions to identify changes among properties. This detection process applies three different measures to identify changes: *number of packages* (NOP), *number of classes* (NOC), and *number of methods* (NOM). Once all changes have been retrieved, a concept lattice is constructed based on the matrix. Meanwhile, a heuristic is used to remove entities which changed more often than the detected concept and to remove concepts with less than two entities. The remaining co-change patterns can then be used for impact analysis as they contain the entities which changed together the most.

Dependencies between source code entities encoded in different programming languages are difficult to determine using traditional impact analysis techniques. Ying et al. [94] propose an approach which is based on mining change patterns among files to suggest possibly impacted source code files to the developer, independent of the programming language. First, their approach extracts historical data from repositories and preprocesses them, by dividing them into sets of atomic changes and by skipping non-useful changes (e.g. changed comments). Then, an association rule mining algorithm is applied to detect reoccurring patterns, using the FP-tree algorithm. All relevant source code files are identified by querying the file under consideration against the mined patterns, to identify files which should change together with this one. Finally, query results are sorted into three groups according to their relevance: *surprising*, *neutral*, and *obvious* based on structural information of the changes.

Hassan and Holt [95] investigate how changes of one entity propagate towards other entities of the same program. They analyze historical development data stored in source code versioning systems by transforming change information (granularity: files) into the more appropriate level of source code entities. The transformation process consists of the classification changes into the groups of changes containing added code entities, and such containing no added code entities. Their approach utilizes several heuristics to capture impacts, namely *historical co-change records*, *code structure relations* (i.e. calls, uses, defines), *code layout* (i.e. location of entities relative to classes and components), *developer data* (entities changed frequently by same developer), *process data* (change propagation depending on the used process), and the *similar names* heuristic. Each heuristic uses several pruning techniques to cut off unlikely results: *frequency pruning* (i.e. remove the least frequent ones), *recency technique* (i.e. return those that where related in the past), and a combination of *frequency pruning* and *recency technique* together with a decay function or a counter. An evaluation of all heuristics revealed that the hybrid ones achieved the best results.

The approach presented by Bouktif et al. [96] proposes files that should co-change with a given entity, based on the detection of change-patterns through *dynamic time warping* (DTW). The approach computes distance measures between two files within a certain time window to filter unrelated files, using distance thresholds. The actual identification of change patterns is a three pronged process, which starts by building an XML-based log file containing all moments in time when files were added, removed, or changed. The second step involves the transformation of data by computing appropriate time windows to limit the effects of past changes, where the length of a window can be determined by either imposing a fixed distance in time ("changes not older than..."), or by imposing a fixed number of past changes ("the last X changes"). Finally, DTW is used to identify change patterns by aligning each possible pair of file histories, and by grouping histories with a distance below a predefined threshold.

Instead of extracting evolutionary data from version control repositories, which involves expensive preprocessing [97], it is also possible to extend IDEs to record changes on the fly. Robbes et al. [98], [99] propose a new concept for a CASE tool supporting software evolution by recording developer activities in an IDE and demonstrate their *SpyWare* platform in [100]. When recording every step of developers, there is no more need for extracting data from repositories, which contain only those information which have actually been committed. The tight integration into the IDE also eases the work for developers, who no longer must cope with a multitude of different tools. Automatically recorded changes enable to model the change history as a sequence of first-class change operations, in contrast to the concept of program versions. There exist a variety of embedded tools within *SpyWare* to facilitate impact analysis and to assist developers when changing software.

9) Combined Technologies: Combining MSR which analyzes a series of versions with traditional impact analysis which is focused on a single version could yield several advantages. Traditional analysis applied on single versions of a program is not adaptive, whereas MSR-based approaches rely on historical data, which might be incomplete or outdated and therefore inferring wrong impacts [101]. Therefore, Kagdi and Maletic [102], [103] are concerned with combining both methodologies to improve the prediction of changes. The basic idea is to apply traditional techniques on single revisions and to infer additional dependencies from version histories using MSR approaches. Using the achieved results to cross validate each other is a potential technique to further improve impact analysis. In [104], Kagdi and Maletic further refine the idea by proposing to convert the source code into an XML representation using *srcML* to enable source code dependency analysis. The authors plan to use the mining tool *sqminer* and the diff-tool *codeDiff* to mine fine-grained co-changes at code level. The last step is focused on matching the granularity of entities obtained through MSR and traditional analysis to incorporate both results by either forming the union (Disjunctive Approach) or intersection (Conjunctive Approach) of both sets.

The approach of German et al. [105] is aimed at identifying parts of a program which have been forgotten to change, by utilizing prior code changes to determine what functions have been modified. The approach is based on the construction of a dependency graph for each function, which contains all invoked functions (similar to a call graph), by defining a window of interest which contains a sequence of changes. This change information is extracted from version control systems by comparing subsequent versions of the program. Each changed node of the dependency graph is marked as *changed*, whereas all successor nodes are marked as *affected*. Unmarked nodes and nodes outside the area of interest are pruned using the following two metrics: *ratio of affected functions*, and *ratio of changed functions*. The result of this process is a *change impact graph* (CIG), which visualizes the impacted elements and can be used for further judging the effects of a change.

Canfora and Cerulo [106] want to combine the information stored in CVS and Bugzilla[3] to improve impact analysis. The authors combine MSR with IR techniques to bridge between both data sources. Therefore, the authors assume that commit reports and previous change requests which focus on a certain file are a good indicator to predict further changes of the same file. The proposed approach computes the similarity between old and new change requests, which is based on textual similarities of bug reports, feature proposals, commit messages, and change requests themselves. The computation of similarities uses standard IR procedures, such as stop word elimination, word stemming, and a probabilistic IR comparison algorithm. Finally, a ranked list of files is presented to the developer listing likely candidates.

---

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

Both authors present a refinement of their approach in [107] to enable more fine-grained analysis. The improved approach recovers the history of source code modifications (added, removed, and changed LOC) from CVS, and creates a line history table, which states when each LOC was modified. Therefore, they are storing each LOC together with related free text (e.g. comments). The association between free text and LOC is established by an indexing process, which generates a ranked list of code entities through IR techniques. The index process is comprised of free text analysis, building document descriptions, and scoring each document description to match documents and code. Free text analysis divides free text into a sequence of index terms, which is compressed by word stemming and stop word elimination. Obtained index terms are counted per document to build the document description, which is then used to describe each source code file together with a change request. Finally, a scoring function is applied to match the document descriptions with related lines of code.

A typical limitation of existing MSR-based techniques is the large amount of data which is spread over a long period of time, which typically complicates the mining of association rules. Ceccarelli et al. [108] propose a new MSR approach which is combined with a statistical learning algorithm to overcome this limitation. The proposed methodology infers mutual relationships between software artifacts and utilizes the Granger causality test to identify consequent changes. Evolutionary dependencies between multiple time series are captured using the Vector Auto-Regression model, whereas the Granger test is used to identify changes which are useful for forecasting future changes of other software artifacts. Canfora et al. [109] extend the work presented by Ceccarelli et al. [108] by performing a more detailed case study, including the analysis of the Firefox web browser. However, the average precision achieved with the Granger test is below 25% in two of four studies, at approximately 25% for one study and 60% for the fourth study, while recall is also between 15% and 60%.

Kabaili et al. [110] investigate how developers can evaluate the changeability of object oriented software, based on a formal model for impact detection proposed by Chaumun et al. [69]. The developed extension of the model encompasses impact analysis and regression testing by analyzing classes, methods, and variables of a program. According to the utilized impact model, the impact of a change depends on the type of the change and the type of relation between two program entities. The process of computing ripple effects begins with identifying directly affected classes. Indirectly affected classes are then computed by analyzing the relations between affected and related classes, while considering the type of change. The identification of affected regression tests to be re-executed is achieved in a similar fashion, based on affected classes.

The contribution of Queille et al. [111] is twofold. First, they separate the definition of impact analysis from program comprehension. Secondly, they develop an impact analysis model which is based on propagation rules and program dependencies. The proposed impact model is based on static source code analysis, i.e. the extraction of a program dependency graph, which is combined with closeness information of textual entities. They define a set of propagation rules for the computation of change impacts, which is based on the dependency graph and is implemented in their *IAS* prototype tool. Barros et al. [112] continue the work of Queille et al. by refining the concept of impact rules and providing steps required for an interactive impact analysis process. The extended rule concept distinguishes between rules which result in a strict impact, and those which result in a potential impact, by assigning a virtual order to potential impacts when recursively propagating changes. Barros et al. enhanced the *IAS* tool by reacting on developer feedback to enable interactive impact analysis, i.e. allowing developers to invalidate proposed impacts and edit the dependency graph.

The goal of the work presented by Huang and Song [113] is to combine static dependencies and dynamic source code analysis to improve the precision of impact analysis. The proposed dynamic approach calculates impacts according to binary dependency information gathered at runtime, by collecting execution traces and considering different entry and exit types (e.g. method entry without passing arguments). The impact of method body changes is then computed as follows. First, the execution-after set is calculated for the changed method. Secondly, the approach constructs a dependency graph for the changed method and adds all related methods to the impact set, according to the relation between them, e.g. *returns into* or *passes parameter*. In contrast, the impact of method interface changes is calculated as follows. First, recorded execution traces are searched for method entry events of the changed method. A backwards trace is then computed to identify the method's caller, and both methods are added to the impact set. Finally, the impacts of atomic changes of attributes are calculated by determining methods which depend on them and adding transitively related methods to the impact set. Huang and Song later improved the first version of their methodology and implemented the approach in the *JDIA* tool [114]. They added support for the identification of runtime inheritance relations, which is achieved by identifying how the inheritance hierarchy is reflected in an execution trace. Therefore, they monitor constructor calls and investigate the differences in event sequences of certain Java-instantiation rules to identify inheritance relationships.

Walker et al. [115] propose a new approach to assess and communicate technical risks, which is based on weakly change estimations, historical change behavior, and the current structure of the software. The approach is built on a probabilistic algorithm, which is combined with dependency analysis and history mining. A structural dependency graph is constructed where each edge is annotated with the probability of propagating changes from the source to the target node. The initial probabilities are computed by extracting atomic change sets from CVS repositories and counting how often two entities changed together. These initial probabilities are then propagated through the graph using a modified version of the Dijkstra-algorithm. Once the graph and all probabilities have been computed, the *TRE* tool presents the graph to the developer who then decides which entity is truly impacted, based on assigned probabilities.

Similar to the work of Huang and Song [113], [114], Maia et al. [116] propose a new approach to combine static and dynamic impact analysis to improve the detection of impacted entities, and to reduce the number of false-positives. The approach combines execution after sequences with the analysis of program dependencies for impact prediction. First, a dependency graph is extracted from source code while using a fixed distance to cut off branches. Secondly, execution after sequences and frequency information are recorded to derive further dependencies (succession relations) for impact analysis. The required frequency information are collected by counting the number of calls per method, as long as the *successor-stop-distance* has not been reached. Finally, results of both steps are joined and ranked based on the impact factor gained in the 2nd step, where the impact factor is the probability of an entity being the successor of an event in the change set, obtained by dynamic analysis.

Wong et al. [117] emphasize the importance of considering the temporal dimension of history when mining change couplings from version history, as the choice of the timespan to be analyzed impacts the achieved results. The authors provide a formalization of logical couplings as stochastic processes, which are based on Markov chains, and define families of stochastic dependencies. The proposed methodology applies a sliding window approach, which equals a discrete k-th order Markow process to limit the length of the historical data which should be analyzed. The probability of dependencies is computed by a smoothing function, which controls how much each transaction contributes to a dependency. This is achieved by preferring recent history over distant history. Stochastic dependencies obtained through this process are then used to reason whether an element will be impacted or not.

By combining information retrieval (IR) and evolutionary couplings for impact analysis, Kagdi et al. [118] provide a developer centric technique which focuses on identifiers, comments, and other source code entities. Their approach obtains conceptual information using IR techniques, and mines evolutionary information (co-change patterns) from version control repositories. The process of identifying possible impacts starts with computing conceptual couplings between entities of the same program version (the most recent version) using IR techniques. Next, evolutionary couplings are mined from version history while not considering the current (most recent) version. The mining process consists of itemset mining, as the order of entities is not important and fine-grained change sets are searched for evolutionary couplings. In a final step, both coupling information are combined to predict the impact. However, it still remains an open question whether to use the intersection or union of both as shown in earlier work of Kagdi and Maletic [104].

Sun et al. [119] propose a static impact analysis technique called *HSMImpact*, which is based on hierarchical slicing models. They apply a stepwise slicing algorithm on four types of hierarchical dependency graphs: *package-level dependency graph* (PLDG), *class-level dependency graph* (CLDG), *method-level dependency graph* (MLDG), and *statement-level dependency graph* (SLDG). The entire HSM analysis process consists of three steps. First, hierarchical change sets are defined at different levels of granularity. Secondly, the slicing algorithm is applied using appropriate slicing criteria for each level of granularity, e.g. variables or classes. Finally, hierarchical impact sets are compute ranging from package to statement level.

The aim of the master thesis of Mohamad [120] is to improve program comprehension by applying impact analysis techniques, where the actual impact analysis is achieved with the help of program slicing and traceability links. The approach of Mohamad is based on the *CATIA* tool developed by Ibrahim et al. [121]–[123]. It supports impact analysis by visualizing program dependencies and traceability links, which are used to trace changes across different entities of the program. The author enhanced the tool by providing a graph view for impacted elements, which replaced the former textual output of *CATIA*.

The PhD thesis of Kagdi [20] is build upon a comprehensive survey of MSR approaches conducted by Kagdi et al. [19], and the work presented by Kagdi and Maletic in [102]–[104]. The author proposes an approach to mine evolutionary couplings from version histories, which is combined with static dependency analysis of single versions to improve coupling detection. The goal of his work is to build a holistic approach for software change recommendation, by validating the outcome of dependency analysis with results achieved by MSR techniques. The mining process is comprised of training and validation phases. The training process mines a certain potion of the version history to infer change couplings between software entities. The validation process then examines extracted couplings to check whether changes were predicted correctly. Apart from impact analysis, his thesis also addresses the application of MSR for mining traceability links from software repositories, and other related tasks.

Lee [124] investigates the additional challenges of ripple effect analysis in object oriented software, which due to its nature and many hidden dependencies, is more complicated than ripple effect analysis in procedural programming languages. The contribution of her PhD thesis is a set of object oriented data dependency graphs, impact analysis algorithms, and evaluation metrics, which have been implemented in the *ChaT* tool. The proposed algorithms, such as *FindEffectInClass* or *FindEffectByInheritance*, rely on the computation of the transitive closure of object *oriented data dependency graphs* (OODG), which are based on control and data flow information extracted from the program. The algorithms analyze relationships among components and weight them according to the type of relation, which is expressed by a set of impact propagation rules.

Similar to the work of Kagdi and Maletic [102], [103], Hattori et al. [14] combine traditional impact analysis and MSR by applying Bayes' theorem to combine results computed with both techniques. First, their approach computes a dependency graph that represents the source code and uses dependency types like *contains*, *isAccessedBy*, and many others. A reachability analysis is then applied on the graph to estimate possibly impacted elements. Next, a MSR approach using a sliding window of 200 seconds is used to extract change sets. Computed change sets and possibly impacted entities are then analyzed by a probabilistic algorithm using Bayes' theorem to remove false-positives, which is based on the frequency of commits. In a final step, the *Apriori* and *Disjunctive Association Rule* (DAR) algorithms are used to weight and sort possible impacts according to their likelihood. As a result of their conducted case study, *DAR* turned out to be more reliable and less resource consuming then *Apriori* when analyzing fine-grained entities.

Many software systems are comprised of different middleware and other COTS components, which communicate via messages. Popescu et al. [125], [126] are concerned with analyzing impacts which spread across system borders due to message communication in event based systems. Their approach analyzes control and data flow to identify incoming and outgoing message interfaces, and distinguishes between inter-component and intra-component dependencies. Inter-component dependencies are collected while analyzing incoming and outgoing component interfaces and matching message types of sinks and sources. In contrast, the intra-component dependencies are identified by calculating a call graph which is annotated with message types and access permission information to analyze the message control flow within a component. Finally, both dependencies are merged into a *system dependency graph* where reachability analysis is used to detect impacted entities.

10) Other Technologies: There are two studies which cannot be classified according to their utilized technology, as one is using a unique method [127], and the other study provides no details on the utilized method [128].

The approach of Moonen [127] introduces a lightweight impact analysis technique, which is based on the concept of island grammars, and provides a reusable and generative framework for impact analysis. The author developed the *ISCAN* tool, which uses a *extract-query-view*-approach and consists of a parser, a repository, and a user interface. The parser utilizes syntactical analysis which is based on island grammars and consist of two parts: the *detailed productions* (the language constructs of interest; the *islands*) and the *liberal productions* (the remainder of the input; the *water*). In this case, the language constructs of interest are COBOL data fields. The remainder of the code, which is not related to these data fields is considered to be *water.* The technique of island grammars brushes aside non interesting parts of the code to speed up the analysis. Every entity which remains in one island after brushing aside the water is reported to be impacted by a change.

Hoffman [128] investigates the complex nature of relationships in object oriented software, which are caused by information hiding, encapsulation, polymorphism, and related concepts. The author concludes that developers must be able to trace such relations, as they support the testing and understanding of software. The proposed approach is based on a multistaged process, the *comparative software maintenance* (CSM) methodology. CSM enables developers to determine the components of a system and their relationships. It allows to model components as *extended low level software architecture* (ELLSA), which creates a virtual software system and can be used for *predictive impact analysis* (PIA). It further allows to compare the ELLSA model structure for PIA and *comparative impact analysis* (CIA) to determine affected components, and to instrument source code for checking test execution coverage. CSM performs change analysis as a base for impact analysis and is implemented in *JFlex*, which scans source files and compiles obtained data into the ELLSA model. This ELLSA model (a copy of original system) can be used to ask "what if" questions (PIA) and enables the comparison of two different versions of ELLSA model.

C. Architectural Models

Existing impact analysis techniques require source code or architectural representations in order to predict changes. However, such data is not always available for all software systems or they are not useful for the person performing the impact analysis (e.g. a requirements engineer might find it hard to analyze source code). Therefore, Aryani et al. [129], [130] propose an approach to predict the change propagation based on information which are visible and understandable to domain users. The approach operates without requiring access to source code or development histories, as it derives information from user manuals and expert knowledge, and stores them in a weighted dependency graph. Logical relations between *domain functions* (functions provided to users), *domain variables* (data with clear meaning at domain level, e.g. a date) and *user interface components* (contains at least one domain function) are used to derive the dependency graph, and enable to reason about change propagation, using reachability analysis.

The work of Briand et al. [131], [132] is concerned with keeping the architecture of a software system in a consistent state and synchronized with the underlying source code. Therefore, the authors propose an impact analysis approach for architectural models, which operates on UML models and supports the entire UML2 specification. The authors use OCL[4] to express explicit rules, which are used to search for impacted elements, where a set of 97 OCL rules is provided in [131]. The propagation of changes to indirectly related software entities is controlled by a distance measure, which is used to either cut off the change propagation or to weight impact paths according to their nesting depth. Another contribution of their work is a taxonomy of change types proposed in [131], which provides three elemental change types: *add*, *remove*, and *change*.

Analysis and design models must change when software evolves. Which elements of design models should change due to a given change, and which elements changed together and are likely to do so again are the motivating questions for the work of Dantas et al. [133]. Their proposed methodology uses historical information to uncover traceability links between UML models, based on mined association rules. Each traceability link can be enhanced with the information *who*, *when*, *where*, *why*, *what*, and *how* it was changed to support the developer. The actual mining of association rules is based on the *Apriori* algorithm which uses support and confidence metrics, and minimum thresholds for pruning proposed impacts.

The work of Xing and Stroulia [134]–[136] is focused on the evolution of UML class diagrams. The motivation for their work is the insufficient support of *diff*-tools for the differencing of logical models. Existing tools are only able to perform a line-based differencing which is not suitable for hierarchical models. However, understanding class evolution is essential for understanding the current design of the software. Thus, they developed the *UMLDiff* tool which provides fine-grained change information between subsequent versions of a class diagram. The tool allows to extract hidden evolutionary dependencies between classes and creates class evolution profiles which can be used to predict future changes.

Current visualization approaches are useful for understanding software evolution, but they assume that evolution of a system can be displayed as an unbroken sequence of time, which does not hold for every system [137]. McNair et al. [137] developed a lightweight approach to examine the impact of changes on a system's architecture by visualizing ripple effects and incomplete histories, which is implemented in the *Motive* prototype. The required information are collected by analyzing a development period within a certain timeframe. This is achieved by preprocessing data from CVS repositories to compute the impact of a change-set by scanning files which were modified by this change. *Motive* then uses these information to display software as a graph and annotates entities as either *added*, *deleted* or *phantom objects* (added and deleted within analyzed timeframe) to visualize the evolution of the system.

Many systems are interconnected with other software components, as they offer special interfaces or services. Changes to one system or its interface are therefore likely to affect connected systems as well. However, it is difficult to estimate the impact of changes, which are spread across system borders. Yoo and Choi [138] propose an XML based approach for *Interface Impact Analysis*, which builds a common information architecture between connected systems to facilitate impact assessment. Their approach translates system specific messages into a more comprehensible and extensible XML format, and constructs a message dependency graph to keep track of system communication and data exchange. Message flow between components is recorded and a interface message filtering algorithm is used to prune paths of messages, which are of no interest for the current change. The obtained message communication graph can then be used to assess the impact of a change on related systems.

Enterprise architectures combine organizational structures, business processes, and the actual software architecture. A single change can affect many different domains of enterprise architectures, since these combined structures are very change prone. Therefore, de Boer et al. [139] analyze and assess impacts on the enterprise-level of an architecture. The proposed analytic process is based on an enterprise architecture modeling language called *ArchiMate*, which is a combination of *Testbed* (a business modeling language) and UML concepts at a very general level. It features a business layer, an application layer, and a technology layer. Architectural entities and dependencies between them are stored in a dependency graph to enable impact analysis, which is based on the tree architectural layers supported by *ArchiMate*.

Vora [140] defines an architectural modeling style called *temporal control flow rule-based architecture* (TeCFRA) and a corresponding ADL called *TeCFRADL*. Both are used to model the design and evolution of software architectures, while supporting non-functional properties at component level. *TeCFRA* consists of an execution and evolution management framework. The execution framework provides external control flow graphs, which allow for flexibility during the analysis and are focused on method invocations. This enables developers to separate applications into activities and tasks, where the later correspond to methods and method calls. Control flow rules are then used to compute the impacts of changes, based on the control flow graphs. The evolution management framework on the other hand captures dependencies between architectural components by mapping evolution specifications to ADL specifications, which contributes to the process of impact analysis.

Analyzing the static structure of component-based software architectures is an important step of architectural impact analysis, which can further be improved by incorporating dynamic UML diagrams, such as sequence diagrams, into the analysis process. Feng and Maletic [141] propose an approach to support dynamic impact analysis on the architectural level by providing a taxonomy of changes

---

[4] http://www.omg.org/spec/OCL/

in component-based architectures, and a set of impact rules to transfer changes among affected components. Their approach derives component interaction traces from component diagrams and sequence diagrams, which are sliced by impact rules to obtain the set of affected entities. The proposed taxonomy for change types distinguishes between atomic changes such as the addition or deletion of an entity and compound changes. The approach is implemented in the *SOCIAT* tool.

The goal of Tang et al. [142] is to record architectural decisions in a structured way, as design rationale is often not documented and therefore getting lost. However, such decisions are a useful source for later impact analysis. Tang et al. use the *architectural rationale and linkage model* (AREL) to capture design decisions and design elements on an architectural level, based on their earlier work on the *architecture rationalization method* (ARM). The authors analyze causal relationships between architectural elements to perform what-if reasoning with elements and relations of the architecture, which are modeled as *Bayesian belief networks* (BBN). The transformation of architectural elements and relations into a BBN is achieved by mapping ARM elements to BBN nodes, and ARM relations to BBN edges. The architect then assigns initial probabilities to elements by making estimations, which are based on personal experiences and architectural assessment. Finally, the BBN and Bayes' theorem are used to compute the likelihood of a change, which also depends on the type of dependency between two entities.

Zhao et al. [143] observed that the affects of changes do not limit themselves on source code, but also spread across architectural models and other design documents. They emphasize the need of addressing the question of impact analysis on an architectural level by expanding tool and method support. The authors propose an approach for automated change effect assessment, which is based on a formal architecture specification using the *WRIGHT* ADL, and define new slicing and chopping methods for software architectures. The approach infers information flow within components from *WRIGHT*-descriptions to construct an architectural flow graph. This graph is then used to compute architectural slices to reveal possibly impacted entities.

Approaches which are trying to predict and assess ripple effects based on historical data alone suffer from incomplete and inconsistent version histories. Therefore, Wong and Cai [101] propose an approach to extract logical models from UML class diagrams and combine them with historical information from software repositories. The proposed approach is based on a logical framework called *augmented constraint network* (ACN), which is used to model design decisions and their relations. UML elements are transformed into ACNs, where relations are added as constraints, and the interface and implementation of a class are added as variables. Impacted elements are identified by weighting classes according to their number of ACNs, where more sub-ACNs result in a higher rank. The distance between two classes also influences the weight, as closely related classes receive a higher ranking. Obtained weights are multiplied with the co-change frequency of entities, which is mined from version histories. Finally, the ten highest ranked elements are reported to the user.

van den Berg [144] proposes an approach for impact analysis, which is based on traceability relations between elements of software artifacts. The author provides formal definitions of specific dependency types, and an algorithm which computes and stores dependencies in matrices. The approach computes traces between dependency matrices of different entities in a double-staged process: the matrices are multiplied and a crosscutting matrix is derived from the obtained product. A crosscutting dependency is defined as a relation where both, source and target node, are involved in other relations as either source or target. The proposed approach also allows for multiple levels of dependencies, by applying a series of matrix operations. Identified crosscutting dependencies are then used to detect possibly impacted elements.

D. Requirements Models

Changing customer needs and constantly changing technology demand for impact analysis on the level of software requirements [145]. A study which investigates the influence of requirements changes in evolutionary development was conducted by Nurmuliani et al. [146], who concluded that late additions of requirements are likely to inflict high costs and long delays in software development. Thus, requirements impact analysis is essential to assess the possible costs and affects of introducing new requirements or of changing existing requirements. Further, a graphical representation of requirements, based on a requirements modeling language, is often considered as helpful for understanding and changing them. However, it is not clear if it really improves the detection of impacts among requirements. Mellegård and Staron [147] conducted an experiment to find out how graphical requirements models influence impact analysis. Therefore, the authors examined differences between traditional text-based requirements and requirements models. The authors use a domain specific modeling language called *graphical Requirements Abstraction Model* (gRAM) for comparing against textual requirements. The *gRAM* language provides four levels of requirements-abstraction: *product*-requirements, *feature*-requirements, *function*-requirements, and *component*-requirements. gRAM further supports three types of traceability-links to connect requirements: *owns*, *satisfies*, and *depends-on*. Eighteen students took part in the experiment, and Mellegård and Staron measured the required effort, achieved accuracy, and perceived confidence while changing requirements, presented either as text or gRAM models. The use of graphical models resulted in considerably decreased analysis time, but slightly decreased accuracy on the other hand.

Only little work has been spent on creating suitable impact analysis techniques for requirements, which is due to their abstract nature. Lock and Kotonya [148] propose a hybrid technique, which combines traceability links and probability estimations for analyzing the impacts of requirements changes. First, traceability links are extracted from the project and combined to a single traceability graph. Each link is further annotated with a probability value, expressing the likelihood that the link will propagate changes. This probability value is based on developer experience gained through case studies, and is a measure for how often this impact path was taken by former changes. The result of this process is the final impact propagation graph, which enables the propagation of changes based on weighted traceability links.

ten Hove et al. [149] investigate the role of traceability for requirements impact analysis and distinguish between the domain (what the stakeholders want to be modeled) and the actual model, which is used by analysts and contains knowledge for certain purposes. Their proposed approach consists of two phases: the validation of external inconsistencies, which define differences between the domain and the model, and the actual model change. External inconsistencies are evaluated by identifying domain changes and splitting them into sets of primitive and atomic changes. A set of impact rules is then applied to propagate changes across the model. The final model change is achieved by mapping and implementing the detected external inconsistencies onto the model.

Hewitt and Rilling [145] analyze scenarios and components, which are stored in *use case maps* (UCM) [23] to assess the impact of requirements changes on a system, where UCM scenarios provide information how system components interact to fulfill the requirements. Their proposed approach is based on dependency analysis among UCM scenarios and components, and operates as follows. First, scenarios are related by common functionality, where functional groups of scenarios are created that carry out the same goal. Secondly, forward and backward dependencies of components are determined and the transitive closure of the obtained dependency graph is computed to enable impact propagation.

Similar to the work presented by Hewitt and Rilling [145], Hassine et al. [150] analyze the impact of requirements changes based on UCM models. They introduce an UCM slicing algorithm to enable early analysis and localization of changes. Therefore, they define dependencies between UCM elements and group them by *functional*, *temporal*, *containment*, and *component* dependencies. The proposed slicing algorithm tries to isolate a set of scenarios and is based on dependencies, which is similar to source code slicing which isolates statements affecting a program variable. The scenario slicing algorithm is implemented in the *CIA* tool, which allows users to decide what they want to change and assess, e.g. adding or removing a function or changing a behavior.

Goknil et al. [151] propose a metamodel for requirements, which is based on well defined requirements relations to enable the tracing of changes for impact analysis. The metamodel contains four types of relations (refine, require, conflict, contain) and supports the addition of new relations as specializations of existing ones. They combine the knowledge of change types (e.g. ”add a requirement”) with the type of relation (e.g. ”A contains B”) to propose possible impacted elements, and provide information how they are impacted (e.g. ”X must be deleted”) by establishing a set of impact rules, which operate on traceability relations.

The automated handling of traceability relations for impact analysis is the driving force of the work of Lee et al. [152]. The authors propose a goal-driven traceability approach for analyzing requirements, which combines the concepts of goals and use cases. The utilized goal concept is very similar to GRL [23], as it distinguishes between *rigid goals* and *softgoals* (GRL contains *goals* and *softgoals*). Goals and use case are connected via three different traceability relations (evolution, dependency, and satisfaction), which are stored in a design structure matrix. Impacted entities can then be determined by applying a reachability analysis on the matrix.

According to the work of Spijkerman [153], requirements are subject to frequent changes right from the beginning, as they are the first artifacts of the software development process. The author exploits the semantics of requirements relations to perform impact analysis by establishing a classification of requirements changes. The contribution of his thesis is a refinement of requirements relations, which are captured by traceability links and a set of impact rules that distinguish between different types of traceability links to facilitate impact analysis.

The PhD thesis of Jönsson [154] explores impact analysis in the light of requirements, and presents organizational views of impact analysis. One central result of his thesis is that most studies focus on technical views of impact analysis, thereby neglecting organizational views, such as helping to understand how impact analysis is actually used in practice. The author further discusses twenty different uses for impact analysis in software development, e.g. analyzing system impacts or time-cost-tradeoffs, and proposes an information retrieval based approach for impact analysis. The proposed IR approach uses *latent semantic indexing* (LSI) in combination with a term-by-document matrix to associate terms with documents. Once the indexing process has been completed, inferred relations can be used for impact analysis.

O’Neal and Carver [155] analyze the affects of changes on already completed work products, and compare requirements changes based on their severity. The authors propose an impact analysis approach, which is based on traceability and a method to prioritize requirements changes based on an impact metric. The impact metric analyzes traceability links between work products and requirements, which are annotated with additional properties. A developer can assign the following properties: complexity of the work product, effort to create the work product in person hours, development phase in which the work product was created, and the influence of the source work product on the target work product. The impact metric is then used to calculate the required effort of changing existing work products, where changes are further grouped into fuzzy compatibility classes by ranking the mean of impact metrics on the classes of a work product.

In later work, O'Neal [156] proposes an predictive approach for requirements impact analysis, which is also based on traceability, the *Trace-based Impact Analysis Methodology* (TIAM). *TIAM* utilizes traceability links and a requirements change impact metric to assess the impact of requirements changes. The proposed methodology operates with different types of changes, each having a different impact, and generates a set of potentially impacted work products for each requirements change. The prediction process operates in different steps and starts with grouping work products by a fuzzy-algorithm. Secondly, the impact metric is applied on work products to calculate their impact values. The obtained set of work products is then ordered according to the impact value to evaluate the risk of implementing a requirements change.

E. Miscellaneous Files and Artifacts

Antoniol et al. [157] focus their effort on extracting co-changing files from CVS repositories. Their approach enables impact analysis on file level, as co-changing files contain non-trivial dependencies, which are otherwise hard to determine. The proposed approach extracts file histories from CVS repositories, according to the time window of interest. Then, *Dynamic Time Warping* (DTW) is used to compute the distance between pairs of histories by computing non-linear mappings of one history to another, and by minimizing the distance between them. Finally, obtained distance measures are used to group and filter similar evolution histories, in order to detect groups of co-changing files and to trace how long they have co-changed.

Clusters of artifacts which frequently change together are good candidates to be grouped in subsystems, and can be used for assessing the impact of changes on single artifacts. Beyer and Noack [158] propose a two-staged methodology for identifying artifacts which should form a subsystem based on their co-change behavior. The authors derive several requirements for co-change layouts, and introduce a model to produce co-change layouts. They extract a co-change graph from version control repositories, which contains vertices (files) and edges (change transactions). The computed layout of the graph reveals clusters of co-changing artifacts, as each edge is labeled with the degree of co-change between two nodes.

Askari and Holt [159] propose three algorithms to predict the likelihood that a certain file will change, where one algorithm also detects co-changing file patterns. The algorithms extract change information from CVS repositories, count the number of modifications for each file, and sort them into a list. This list is used to create a sequence of file changes which based on the *Most Likely Estimation* (MLE) model, can be used for change prediction. The 2nd algorithm enhances the concept of MLE with *Reflexive Exponential Decay* (RED), which decreases the rank of each file periodically if the file is not modified again. The 3rd algorithm, *RED-Co-Change* (REDCC), is an improvement of RED which integrates co-change information into the prediction process by updating the rank when co-changing files are modified. Developers concerned with impact analysis can benefit from the ranked list, as the highest ranked files are worth to be inspected when changing the software.

Most impact analysis techniques are limited to source code, and do not consider non-source files. Thus, Sherriff and Williams [160] propose an impact analysis approach which is able to analyze any type of file, and is not restricted to source code files only. Their approach extracts change records from a repository, compiles them into a matrix, and computes association clusters through singular value decomposition. Each file of a cluster is weighted according to its degree of participation in the cluster. High singular values are a good indicator that a file will be affected by changes to other files of the same cluster.

Jashki et al. [161] share the same motivation as Sherriff and Williams [160], and extend impact analysis to files other than source code files. Their approach analyzes change logs from software repositories to derive mutual file manipulations, which are then stored in a matrix. Secondly, *Principle Component Analysis* (PCA) is applied to reduce the dimension of the matrix in order to reduce the complexity of further analysis steps. Finally, clusters of closely related files are identified by applying five standard clustering algorithms (K-means, X-means, Spectral clustering, EM, DBscan) on the reduced matrix.

Configuration Management Databases (CMDB) provide an alternative to version control repositories such as CVS or SVN when mining change-couplings. Nadi et al. [162] extract historical change data from a CMDB system, and use support and confidence measures to assess how closely two entities are related. In this context, support states how often two entities changed together where confidence is the ratio of co-changes compared to the total amount of changes of one entity. Both measures are stored in separate matrices and enable a transitive reachability analysis. The reachability analysis is supported by three different pruning techniques: *minimum support*, *minimum confidence*, and *exponential forgetting*, which utilizes the concept of half-life time to forget about coupling information from changes which took place long ago.

F. Combined Scopes

The work of Kim et al. [163] is focused on impact analysis for product-line development in huge companies, which requires assessment of source code and architectural models once basic components have changed. The proposed impact analysis approach for software architecture and source code is build on the concept of reflexion models, as proposed by Murphy et al. [164], [165]. The simplified reflexion model is computed as follows. First, a hypothesized high-level architectural model is defined, which is then mapped to the extracted source code model, using mapping rules. The resulting model is then analyzed using dependency and reachability analysis. The proposed process has been implemented in the *iCIA* tool, which is capable of visualizing impacts and the computed reflexion model.

An approach which determines, whether code changes impact the system's architecture is proposed by Hammad et al. [166]. Their aim is to keep the architecture synchronized with source code, therefore applying impact analysis on both levels. They distinguish between code changes that affect the design (e.g. "add a method"), and such which do not affect the design (e.g. "change a control loop"). Their approach ignores changes that do not affect design, which is achieved by applying the tool *srcTracer*. The tool transforms source code into XML using the *srcML* tool, and identifies changes by applying the *srcDiff* tool on the XML output. Rules encoded as XPath queries are then used to identify changes that impact the design. For example, if a rule detects a class in code which is not represented in design models, a design change took place and there is an impact on the architecture.

Sharafat and Tahvildari [167], [168] propose a probabilistic approach to enable the change prediction of classes, which is based on analyzing UML class diagrams and Java source code. Their approach combines dependencies which have been extracted from UML class diagrams and source code metrics. They distinguish between internal and external dependencies, where internal dependencies are computed with source code metrics, and external dependencies require comprehensive analysis. Cyclic external dependencies are resolved by either applying systems of linear and non-linear equations [167] or by using depth-first-search [168]. Once all dependencies have been resolved, the probability of internal changes and the probability of propagating external changes is computed based on Bayes' theorem. However, probabilities must still be normalized, as the authors assume that internal changes which took place in different development periods are independent of each other.

The use of COTS components adds new challenges to the maintenance of software, as the black-box nature of such components decreases the usefulness of most impact analysis approaches. However, without proper analysis of COTS components, software is hard to maintain and adapt to changing requirements. Therefore, Hutchinson et al. [169] define a process for COTS-based software development: *component oriented software engineering method* (COMPOSE). *COMPOSE* is comprised of an ADL (*CADL*) and a model for impact analysis, which is based on the ADL. The system's architecture is described in *CADL* and linked with requirements via traceability relations. Based on dependencies expressed as traceability links, they conduct a simple reachability analysis on the *CADL* graph to determine possibly impacted entities. Kotonya and Hutchinson [170] extend *COMPOSE*, as presented in their earlier work [169]. *COMPOSE* is transformed into a cyclic process, which includes a verification of the software after each step. An additional intermediate layer is added to *CADL* to facilitate the mapping between components and requirements. Furthermore, they distinguish between *directly affected* (connected to a changed entity), *secondary impacted* (connected via *CADL* with an impacted element), and *peripheral impacted* entities.

Business processes undergo many changes, which must be reflected on the source code of the underlying business application. However, mapping such changes to source code is not trivial. Therefore, Xiao et al. [171] investigate impact analysis in the scope of business processes in service oriented business applications. Their approach combines the analysis of requirements encoded in BPEL, and source code analysis via call graphs. Data and control dependencies between BPEL elements are analyzed by a set of impact rules to determine the impact of a change, while a call graph is constructed from the underlying source code. Impacts of BPEL elements are mapped to methods, which then act as starting impact set for source code impact analysis. Changes are then propagated through the call graph, where a distance metric is used to estimate the likelihood of impacts.

The work of Bohner [37], [172] is concerned with impact analysis in systems, which are composed of existing solutions, as more software is build upon middleware and other preexisting components. The author proposes the use of a dependency graph and a dependency matrix, which are combined with a distance measure to limit the impact propagation. The author further discusses the use of XML to facilitate impact analysis in COTS software. Bohner and Gracanin [173] extended this work by focusing on possibilities for 3d visualization of impact relations and dependencies between software entities.

Khan and Lock [174] are concerned with tracing requirements to architecture, and utilizing these traces for impact analysis. They investigate, whether concern-based dependencies help to identify unstable components and anticipate changes. A taxonomy of dependencies is established as the basis of their proposed impact analysis approach. Dependencies are divided into *goal*, *task*, *service*, *conditional*, *infrastructural*, and *usability* dependencies, which can overlap, intertwine or confirm to each other. Their approach uses concentration metrics to measure how often entities are connected via a certain dependency type to evaluate their likelihood for being impacted.

Yu et al. [175] propose the concept of requirements change probabilities to estimate whether a change to one component will spread to other components. Architectural components are extracted and compiled into a $N \times N$ matrix, which is used to store the propagation probabilities between pairs of components. The required probabilities are predicted by applying three metrics on each component:

backward functional call dependency (How many functions of this component are called by other components?), *forward functional call dependency* (How many functions from other components are called by this component?), and *total functional call dependency* (both combined). Impacted components can then be determined by applying a minimum probability threshold.

When changing software, it requires testing to verify the correctness of implemented changes. However, executing all test cases again after each change is not feasible in practice. Regression testing is required to identify those tests which are affected by a change. Briand et al. [176] propose an approach for regression test selection, which is based on impact analysis of software architectures modeled in UML. They distinguish between three classes of tests: *reusable* (the test is valid), *retestable* (the test is valid, but must be rerun), and *obsolete* (the test can no longer be used). The software architecture is connected with regression tests through traceability links, and impacts are propagated across the traceability relations. The impact estimation and selection of tests is based on identifying differences between two versions of the system architecture, where UML sequence diagrams are used to identify impacted test cases.

von Knethen and Grund [177] emphasize the usefulness of traceability links for software maintenance. However, there remain of couple of open questions, which limit the applicability of traces for impact analysis: which kinds of traces should be established, who has to establish them, and who has to analyze them? To answer these questions, they investigate impact analysis based on different stakeholder roles: *project planner*, *requirements engineer*, and *developer*. They developed a tool suite which allows different stakeholders to view traces and execute different analysis steps, where two components are of special interest for developers when performing impact analysis. The component *RelationFinder* searches entities for traceability relations based on similarities of textual attributes. The *RelationViewer* orders traces according to the type of the relation, and displays them as tree. The developer then has to decide which related entities are affected based on an assessment of identified traces.

Connecting requirements, source code, and test cases via traceability links enables comprehensive impact analysis, as proposed by Ibrahim et al. [121]–[123]. The authors developed an approach which gathers traceability relations from different sources. Requirements and test cases are connected while analyzing the system documentation. Test cases and methods are linked via test execution, where methods and classes are linked by static program analysis. In [121] and [122], the authors propose three traceability detection techniques: *explicit links* (dependency analysis), *name tracing* (IR-based), and *concept location*. Ibrahim et al. [123] added a 4th technique called *cognitive links* [178], which enables designers to specify traces manually to improve impact detection. The result of this multistaged process is a dependency graph spanning all available artifacts, which can then be used for propagating changes and determining impacted elements based on reachability analysis.

Looman [179] highlights that requirements are the driving force when developing a system, where architecture implements the requirements and both undergo constant changes. The author proposes an impact analysis process for software architectures, which is based on functional requirements. Functional requirements are transformed into formal behavior descriptions that state which requirements should be present or absent in the current architecture. Architectural entities are then related with corresponding requirements via traceability links, to enable impact analysis by evaluating behavior descriptions. A failed behavior description indicates that there is a change impact between the requirement and the linked architectural component.

Bridging between architecture and source code to allow for impact analysis on both levels is the goal of the work presented by Hassan et al. [180]. The authors propose the *architectural software components model* (ASCM), which is able to express architectures that have been modeled in different ADLs, and the *software component structural model* (SCSM) to couple source code with ASCM models. The approach is based on an expert system, which provides automated impact rule management to cope with software evolution. The utilized expert system consists of a fact base containing the ASCM architecture, a rule base which contains propagation rules, and the inference engine which applies the rules on the fact base. The utilized rules are able to add and remove entities, and consist of preconditions and invariants. An entity is impacted, if a precondition is met and at least one of its invariants is violated. The detected impact is then propagated to its neighbors according to the incoming and outgoing relations between the entities. The connection between architecture and source code is established by linking ASCM entities via projection relationships to corresponding SCSM entities, which enables the propagation of impacts between both models.

IV. RESULTS OF THE REVIEW

The following section discusses the results of the review regarding the goal of reviewing and classifying impact analysis approaches (Section IV-A), evaluating the taxonomy (Section IV-B), and identifying open research questions and future work (Section IV-C).

A. Classification of Approaches

Section III-B to Section III-F of this article are concerned with classifying and exploring approaches proposed for impact analysis. The motivation and methodology of 150 approaches were described to summarize their main idea. The work presented here can therefore be used as a starting point for exploring the field of impact analysis. Furthermore, we classified the 150 approaches according to the criteria of our taxonomy [9]. Table II presents the results of this classification process.

The classification revealed that 65% of all studied literature is concerned with analyzing source code changes and the impact on source code. Only 11% analyze changes and their impacts on software architectures, another 7% analyze requirements. The impacts of changes on other software development artifacts, such as documentation or configuration files, are investigated by 4% of all approaches. Finally, 13% include several kinds of artifacts in their analysis process, e.g. source code and architecture. Figure 2 summarizes these findings.

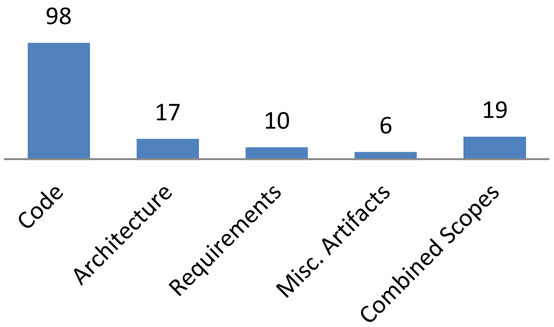

Fig. 2. Amount of approaches ordered by the *Scope of Analysis* criterion

B. Evaluation of our Taxonomy

A problem of existing classification schemes, such as the ones proposed by Arnold and Bohner [7] or Kilpinen [21], is the lack of practical evaluation, i.e. demonstrating that the proposed classification is applicable on reviewed literature. The evaluation of our taxonomy and its criteria was done by checking the coverage of each single criterion. The results of this coverage analysis are depicted in Figure 3 and will be discussed in the following section for each criterion.

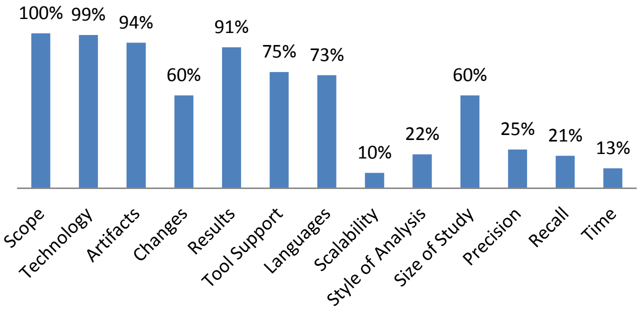

Fig. 3. Coverage of criteria in literature

The criterion Scope of Analysis could be retrieved from all studied literature, as the aim of an approach is easy to identify. The utilized technology could also be determined from all but one approach and proposed ideas were clustered into the ten groups as proposed in [9]. Details about the analyzed artifacts and computed results are provided by more than 90% of all studies, whereas only 60% mention supported change types. However, this information might be required if a developer has to choose an approach.

More than 70% of all studies provide information whether the approach has been implemented (and by which tool) and what programming or modeling languages are supported by the analysis process. In contrast, only 22% of all studies provide any information about how their proposed approach can be used for impact analysis in practice, i.e. whether their approach is *exploratory*, *search-based* or simply analyzes the whole software at once (global). Likewise, even less studies (10%) provide detailed scalability estimations in terms of time and space complexity. This is most likely caused by the complex notion of such estimations in Bachmann-Landau notation. However, studies which contain no evaluation by case studies should at least provide scalability estimations to be comparable to other approaches.

The additional criterion of evaluation results is covered by 67% of all studies and therefore fulfills its purpose of assisting with comparing approaches. 60% of all examined studies provide information about the size of the conducted case study or experiment in terms of LOC, the number of classes, and similar measures. 25% of all studies report on achieved precision, 21% on achieved recall, and 13% on required computation time. Further details on memory consumption were provided by two studies only, so we excluded this as a criterion due to the lack of coverage by 99% of the literature.

Our main criteria (scope of analysis, utilized technology, granularity of artifacts/changes/results, tool support, supported language) are covered by 85% of all approaches in average. The auxiliary criteria (scalability, style of analysis, size of study, precision, recall, computation time) are covered by 25% of all approaches in average. Therefore, we conclude that our taxonomy in its current state is useful and applicable for classifying, comparing, searching, and evaluating impact analysis approaches based on its criteria.

C. Identification of Future Work

Based on the performed validation of the taxonomy, which revealed that 33% of all proposed approaches are not yet evaluated, we investigated how many approaches of a certain scope lack any validation. Figure 4 shows that 80% of all requirements approaches, and more than 50% of architectural and combined approaches were not evaluated by case studies or experiments. Also, as revealed by Table II, many approaches were evaluated with rather small case studies and therefore actually prevent any generalization of results. One possible solution to overcome this limitation would be to establish a universal benchmark, which provides a set of different artifacts, ranging from requirements specifications to source code, to provide a common database for evaluating approaches. However, this is not feasible in practice. Instead, a set of evaluation guidelines could be developed, i.e. stating which measures (e.g. precision and recall) should be used. This guideline could also contain clusters for classifying a system based on its size and complexity, which would allow for better comparison of results.

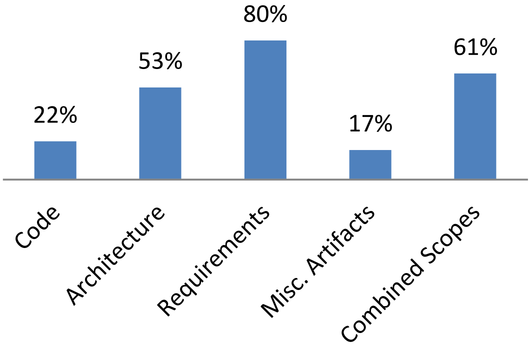

Fig. 4. Amount of approaches not yet evaluated, ordered by *Scope of Analysis* criterion

Another possible aspect for further research is to overcome the lack of approaches which span the entire software development process. Impact analysis is required in the early stage of requirements capturing, in the phase of architectural design, and the final implementation and maintenance phases. However, only 13% of all approaches combine two or more scopes in their analysis process as shown by Figure 2. Therefore, we advocate that more attention should be paid on linking requirements, architecture, and code to enable comprehensive impact analysis. Figure 5 illustrates the amount of combined approaches per scope. Also, according to the work of Kilpinen [21] more information are required how to incorporate a proposed approach into the actual maintenance or development process. This flaw is also supported by the fact that only 22% of all studies reported on the interactivity of the proposed approach, as illustrated by the coverage of the Style of Analysis criterion in Figure 3.

We also noticed that many authors (e.g. [33], [43], [71], [76], [85], [141]) provide a taxonomy of change types to facilitate the identification of impacts. However, the majority of established classifications is not based on a review of related studies on change types, such as the work of Baldwin and Clark [181] (pp. 132-142). We are convinced that more work should be spent on creating a taxonomy of change types for source code, architectural models, requirements, and other artifacts which can be integrated into impact analysis.

In a similar fashion, many approaches are build on a classification of dependency types (e.g. [29], [125], [126], [174]) and suffer from the same problem. The proposed dependency types can be mapped to a set of elemental dependencies which should be defined by a taxonomy. As suggested for change types, we emphasize the need for a more thorough investigation of dependency types in the scope of impact analysis.

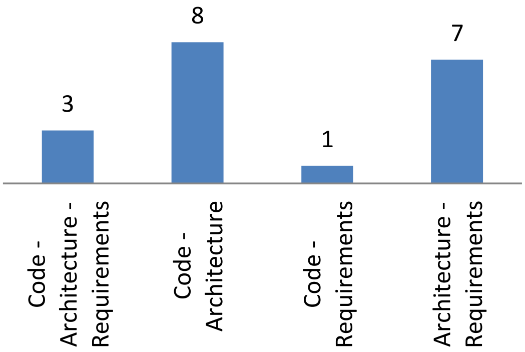

Fig. 5. Combined approaches, ordered by their supported *Scope of Analysis*

V. CONCLUSION

The evolution of software systems and ongoing changes demand for explicit means to assess the impact of a change on existing artifacts and concepts. Thus, software change impact analysis is in the focus of researchers in software engineering. As a result, the amount of studies published in the field of impact analysis is vast. However, there is no extensive review on published literature, which could be used as a starting point for further investigations of impact analysis.

We presented a comprehensive review of impact analysis, which includes the analysis of 150 approaches and related, auxiliary literature. All 150 studies were classified according to the referred taxonomy to provide a tabular overview of the field.

We evaluated our taxonomy by checking the coverage of its criteria in practice, which revealed a coverage of 85% for the main criteria, and 25% for the secondary criteria. Thus, the taxonomy is applicable in practice and assists with the tasks of classifying, searching, comparing, and evaluating impact analysis approaches.

The review also identified a series of open research questions and opportunities for further investigations. First, the review revealed a lack of empirical validations of proposed ideas, which can be quantified as 33% of all studies. Secondly, there is still a lack of approaches spanning the entire software development process, which requires a tighter coupling between different analysis phases. Finally, many authors propose classification schemes for change types and dependency types, which influence impact analysis. However, a more systematic investigation is required, which also compares and incorporates existing work.

Approach	Scopes	Techniques	Granularity of Entities			Tool Support	Supported Languages	Scalability	Style of Analysis	Experimental Results			
			Entities	Changes	Results					Size	P	R	Time
Ryder and Tip [4]	Code	CG	class, method, variable, test case	+/- class, +/-/chg. method, +/- variable	test case	no	Java	-	-	-	-	-	-
Ren et al. [26]–[28]	Code	CG	class, method, variable, test case	+/- class, +/-/chg. method, +/- variable	test case	Chianti	Java	-	Expl.	123 kLOC, 11k changes	-	1.0	10 min
Xia and Srikanth [30]	Code	CG	statements	-	statements	no	-	-	-	-	-	-	-
Badri et al. [31]	Code	CG	method	chg. method	method	PCIA Tool	Java	-	-	77 classes	-	-	-
Briand et al. [32]	Code	DG	class	-	class	Concerto2/ AUDIT	C++	-	-	40 kLOC	-	-	-
Kung et al. [33]	Code	DG	class, method, variable	+/-/inh./vis. class, +/-/sig./vis. method, +/-/typ./vis. variable	class, method, variable	OOTME	C++	-	-	$>$ 140 classes	-	-	-
Li and Offutt [43]	Code	DG	class, method, variable	+/- class, +/-/sig./vis. method, +/-/typ./val./ vis. variable	class	no	-	T: $O(m^3n^2)$	-	-	-	-	-
Rajlich [34]	Code	DG	class	+/-/chg. class	class	Ripples 2	C, C++	-	Expl.	2 kLOC	-	-	-
Pirklbauer et al. [35]	Code	DG	-	-	-	CIAMSS	COBOL	-	-	-	-	-	-
Zalewski and Schupp [38]	Code	DG	STL spec.	+/- STL spec.	STL spec.	no	C++	-	-	-	-	-	-
Petrenko and Rajlich [39]	Code	DG	class, method, statement, variable	-	class, method, statement, variable	plug-in for JRipples	Java	-	Expl.	550 kLOC	$<$ 0.19	-	-
Black [40]	Code	DG	variable	-	variable	REST	C	-	-	725 LOC	-	-	-
Lee et al. [6]	Code	DG	class, method	-	class	ChAT	-	-	-	30 kLOC	-	-	-
Beszédes et al. [44]	Code	DG	class, method	-	class	-	C++, Java	T: $O(n{*}e{+}n*k*m)$	-	400 classes	0.85	1.0	-
Bilal and Black [42]	Code	DG	class, method	-	class, method	REST, CodeSurfer	C++	-	-	-	-	-	-
Jász et al. [45]	Code	DG	method	-	method	CodeSurfer	C, C++, Ada	T: $O(n{+}e)$	-	1.4 mLOC, 83k methods	0.87	1.0	3h
Chen and Rajlich [46]	Code	DG	method, variable	-	method, variable	RIPPLES	C	-	Expl.	-	0.09	1.0	-
Gwizdala et al. [47]	Code	DG	class, method, variable	-	class	JTracker	Java	-	Expl.	400 classes	-	-	-
Bishop [48]	Code	DG	class, method, variable	+/- method	class	Incremental Impact Analyzer	Java	-	-	9 KLOC	-	-	350ms

Fasching [36]	Code	DG	-	-	-	CIAMSS	-	-	Gloabl	6k artifacts	-	-	-
Tonella [53]	Code	SL	variable	chg. statement	variable	reachability tool	C	-	-	2 kLOC	-	1,0	-
Korpi and Koskinen [50]	Code	SL	variable	-	variable	GRACE	Visual Basic	-	-	18.7 kLOC	-	-	5 s
Vidács et al. [55]	Code	SL	macro	chg. macro definition	class, method, variable, macro	Columbus C/C++ frontend	C, C++	-	Search-based	-	-	-	-
Binkley and Harman [54]	Code	SL	variable	-	variable	CodeSurfer	C	$O(n^2)$	Search-based	179 kLOC	-	1.0	-
Gallagher and Lyle [51]	Code	SL	variable	+/-/val. variable	variable	-	-	T: $O(n^2 eloge)$	-	-	-	-	-
Hutchins and Gallagher [52]	Code	SL	variable	val. variable	variable	Surgeon's Assistant	C	-	Search-based	-	-	-	-
Korpi and Koskinen [50]	Code	SL	variable	-	variable	GRACE	Visual Basic	-	-	18.7 kLOC	-	-	5 s
Binkley and Harman [54]	Code	SL	variable	-	variable	CodeSurfer	C	$O(n^2)$	Search-based	179 kLOC	-	1.0	-
Santelices and Harrold [56]	Code	SL	statement	chg. statement	statement	DUAForensics	Java	-	Search-based	21 kLOC	-	-	2 h
Apiwattanapong et al. [62]	Code	ET	method	chg. method	method	EAT	Java	T: $O(n)$ S: $O(n)$	-	33 kLOC	0,24	-	-
Breech et al. [60]	Code	ET	method	chg. method	method	DynamoRIO, RVM	Java, C++, C, Fortran	-	Search-based	131 kLOC	-	-	4 h
Law and Rothermel [57]	Code	ET	method	chg. method	method	Codesurfer	C	T: $O(n)$	Search-based	$>$ 6 kLoc	-	-	-
Law and Rothermel [58]	Code	ET	method	chg. method	method	-	C	T: $O(n)$ S: $O(n)$	-	$>$ 6 kLoc	-	-	58 min
Orso et al. [59]	Code	ET	method	chg. method	method	JABA	Java	-	-	60 kLOC	-	-	-
Breech et al. [61]	Code	ET	method	chg. method	method	-	C	T: $O(n^3)$ S: $O(n^2)$	-	40 kLOC	-	-	38 min
Gupta et al. [65]	Code	ET	variable	val. variable	variable	-	-	-	-	-	-	-	-
Gupta et al. [66]	Code	ET	method, statement	+/-/chg. method, +/-/chg. statement	method	-	-	-	-	-	-	-	-
Huang and Song [63]	Code	ET	method	+/- method	method	no	-	-	-	-	-	-	-
Vanciu and Rajlich [67]	Code	ET	method	-	method	Reveal	-	T: $O(s * T + s * n^2)$	-	190 classes, 1.6k methods	0.543	-	5h
Beszédes et al. [64]	Code	ET	method	-	method	JImpact	Java	T: $O(n)$ S: $O(m*n)$	Global	2,3 kLOC	0.35	-	-
Chaumun et al. [69]	Code	ER	class, method, variable	+/-/vis. class, +/-/sig./vis. method, +/-/vis./typ. variable	class, method, variable	-	C++	-	-	$>$ 1k classes	-	-	-

Sun et al. [71]	Code	ER	class, method, variable	+/-/inh./mod./vis. class, +/-/mod./vis. method, +/-/mod./ vis. variable	class, method, variable	JHDG	Java	-	-	157 classes	0.541	0.712	-
Han [68]	Code	ER	module, class, method, statement	+/-/inh. class, +/-/chg. method	class, method	-	C++	-	-	-	-	-	-
Arisholm et al. [70]	Code	ER	class	-	class	JDissect	Java	-	-	17 kLOC, 408 classes	-	-	-
Poshyvanyk et al. [75]	Code	IR	class, method	-	class	IRC^{2}M	C++	-	-	4 mLOC	$<$ 0.28	$<$ 0.66	-
Vaucher et al. [74]	Code	IR	class	+/-/chg. method	class	PTIDEJ	Java	-	-	790 classes	-	-	-
Antoniol et al. [73]	Code	IR	-	-	-	-	C++	-	-	-	0.487	0.696	-
Zhou et al. [76]	Code	PM	class, method, variable	+/-/m. class, +/-/m. method, +/-/m./val. variable	class, method, variable	Evolizer	Java	-	Expl.	-	0.815	0.623	-
Tsantalis et al. [77]	Code	PM	class, method	inh. class, +/- method	class	yes, unnamed	Java	-	Global	169 classes	0.554	-	-
Abdi et al. [78], [79]	Code	PM	class, method, variable	+/-/vis. class, +/-/vis. method, +/-/vis. variable	class	PTIDEJ	Java	-	-	394 classes	0.689	-	-
Abdi et al. [80], [81]	Code	PM	class	-	class	BNJ	-	-	-	-	-	-	-
Mirarab et al. [82]	Code	PM	adaptable	adaptable	adaptable	Smile and other, not named tools	Java	-	-	263 kLOC, $>$ 6k revisions	0.63	0.259	-
Gethers and Poshyvanyk [83]	Code	PM	class	-	class	-	C++, Java	-	-	1.9 mLOC	0.118	0.446	-
Hassan and Holt [95]	Code	HM	class, method, variable	CVS record	class, method, variable	-	C	-	-	$>$ 15k revisions	0.51	0.49	
Ying et al. [94]	Code	HM	source file	CVS record	source file	-	C++, Java	-	Global	$>$ 20k files, $>$ 100k revisions	0.4	0.2	55 min
Kagdi [103]	Code	HM	class, method, statement	change record	class, method, statement	sqminer, srcML, dwdiff, codeDiff	-	-	-	-	-	-	-
Gall et al. [86]	Code	HM	class	CVS record	class	-	Java	-	Global	500 kLOC	-	-	-
Zimmermann et al. [87]	Code	HM	method, variable	+/-/chg. method, +/-/val. variable	method, variable	ROSE	Java, C++, C, Python	-	Expl.	$>$ 34k files, $>$ 53k revisions	0.38	0.416	-
Gîrba et al. [92]	Code	HM	class	+/- method	class	Van, Moose	Smalltalk	-	Global	$>$ 500 revisions	-	-	-
Gîrba et al. [93]	Code	HM	package, class, method	+/-/chg. method, +/-/chg. statement	package, class, method	-	-	-	-	281 kLOC	-	-	-

Bouktif et al. [96]	Code	HM	source file	CVS record	source file	no	Java, C++, C	-	Global	$>$ 9k files	0.772	0.792	3 min
Robbes and Lanza [98]	Code	HM	package, class, method, statement, variable	+/- package, +/-/inh. class, +/- method, +/-/val. variable	-	SpyWare	Smalltalk	-	-	-	-	-	-
Robbes et al. [99]	Code	HM	package, class, method, statement, variable	+/- package, +/-/inh. class, +/- method, +/-/val. variable	-	SpyWare	Smalltalk	-	-	40 classes	-	-	1 min
Robbes and Lanza [100]	Code	HM	package, class, method, statement, variable	+/- package, +/-/inh. class, +/- method, +/-/val. variable	-	SpyWare	-	-	-	-	-	-	-
Fluri et al. [84]	Code	HM	class, method, variable	+/-/chg. class, +/-/chg. method, +/-/chg. variable	class, method, variable	-	Java	T: $O(n^2)$	Global	26 kLOC	-	-	-
Fluri and Gall [85]	Code	HM	class, method, statement, variable	+/-/inh./mod./vis./r. class, +/-/sig./mod./vis./r. method, +/-/chg. statement, +/-/vis./mod./typ./r. variable	class, method, statement, variable	ChangeDistiller	Java	-	-	1.4k classes	-	-	-
Popescu et al. [125]	Code	MDG, SL	component	-	component	Helios	Java, C#, C++	-	-	19 kLOC	-	-	-
Popescu [126]	Code	MDG, SL	component	-	component	Helios	-	-	-	19 kLOC	-	-	-
Kagdi and Maletic [102], [104]	Code	HM, DG	adaptable	change record	adaptable	sqminer, srcML, dwdiff, codeDiff	-	-	-	-	-	-	-
Ceccarelli et al. [108]	Code	HM, PM	source file	CVS record	source file	-	-	-	-	$>$ 10k revisions	0.8	$<$ 0.3	-
Canfora et al. [109]	Code	HM, PM	source file	CVS record	source file	-	Java, C, C++	-	-	$>$ 500 files, $>$ 1.7k revisions	0.31	$<$ 0.6	-
Wong et al. [117]	Code	PM, HM	-	-	-	-	Java	-	-	278 kLOC	0.618	0.355	-
Hattori et al. [14]	Code	DG, HM, PM	class, method, variable	+/-/vis./inh. class, +/-/vis. method, +/-/vis. variable	class, method, variable	Impala	Java	-	-	3.6 kLOC	0.875	0.775	-
German et al. [105]	Code	DG, HM	method	rename, merge, split, clone	method	-	C	-	-	-	-	-	-
Kabaili et al. [110]	Code	DG, ER	class, method, variable	+/-/inh. class, +/- method, +/- variable	class	-	C++	-	-	-	-	-	-
Canfora and Cerulo [106]	Code	HM, IR	source file	-	source file	yes, unamed	-	-	-	$>$ 1.4k files	$<$ 0.36	$<$ 0.67	-
Queille et al. [111]	Code	DG, ER	-	-	-	IAS	C, C++	-	-	2 kLOC	-	-	-

Barros et al. [112]	Code	DG, ER	-	-	-	IAS	-	-	-	-	-	-	-
Huang and Song [113]	Code	ET, DG	method, variable	+/- method, +/-/val. variable	method, variable	no	Java	-	-	-	-	-	-
Huang and Song [114]	Code	DG, ER, ET	class, method, variable	+/-/inh. class, +/- method, +/-/val. variable	class, method, variable	JDIA	Java	-	-	903 kLOC	-	-	103 s
Walker et al. [115]	Code	DG, PM, HM	type	CVS record	type	TRE	Java	T: $O(nlogn)$	Global	-	-	-	-
Maia et al. [116]	Code	DG, ET	class, method, variable	+/- class, +/- method, +/- variable	-	SD-Impala	Java	-	-	6 kLOC	0.274	0.566	-
Kagdi et al. [118]	Code	IR, HM	statement	CVS record	statement	srcML, srcDiff, sqminer	C++, C, Java	-	-	367 kLOC, 2k files	0.852	0.455	-
Sun et al. [119]	Code	SL, DG	package, class, method, statement, variable	+/- package, +/-/r. class, +/-/sig./r. method, +/- /r./t. variable	package, class, method, statement, variable	JHSA	Java	-	-	3.9 kLOC	0.184	-	-
Buckner et al. [182]	Code	-	class	-	class	JRipples	Java	-	Expl.	-	-	-	-
Mohamad [120]	Code	TR, SL	package, class, method	-	package, class, method	CIA-V	C++	-	Expl.	4 KLOC	-	-	-
Kagdi [20]	Code	DG, HM	file, class, method, variable	-	file, class, method, variable	codeDiff, sqminer	-	-	Global	600 KLOC	-	-	-
Lee [124]	Code	DG, ER	class, method, variable	+/-/inh. class, +/-/vis./r./sig. method, +/-/typ./val. variable	class, method, variable	ChaT	C++	-	Global	29 KLOC	-	-	-
Ren [29]	Code	CG, ET	class, method, variable	+/- class, +/-/sig. method, +/- variable	test case	Chianti	Java	-	Global	123 KLOC, 700 classes, 7k methods	-	-	-
Canfora and Cerulo [107]	Code	HM, IR	source file, statement	+/-/chg. statement	source file, statement	Jimpa	Java, C++	-	Global	272 kLOC, 1.5k files	$<$ 0.15	$>$ 0.7	400 s
Hoffman [128]	Code	-	class, method	-	class, method	JFlex	Java	-	Global	-	-	-	-
Moonen [127]	Code	Island Grammar	variable	-	variable	ISCAN	COBOL	-	-	901 kLOC	-	-	26 min
Aryani et al. [129]	Arch.	DG	component	-	component	-	-	-	-	-	-	-	-
Aryani et al. [130]	Arch.	DG, PM	domain var., domain func., UI comp.	-	domain var., domain func., UI comp.	-	-	-	-	104 kLOC	0.614	0.428	-
Briand et al. [131], [132]	Arch.	ER	entire UML	-	entire UML	iACMTool	UML	-	-	-	-	-	-

Dantas et al. [133]	Arch.	TR, HM	entire UML	-	entire UML	Odyssey-SCM	UML	-	-	60 kLOC	0.6	0.15	-
Xing and Stroulia [134], [135]	Arch.	HM	class	+/-/r./m. class	class	JRefleX	UML	-	-	144 classes	-	-	-
Xing and Stroulia [136]	Arch.	HM	package, class, interface, variable	+/- package, +/-/r./m./vis. class, +/-/r./m./vis. interface, +/-/r./m./vis. variable	package, class, interface, variable	JDEvAn	UML	-	-	800 classes	0.955	-	58 min
McNair et al. [137]	Arch.	HM	component, package, class	+/-/chg. component, +/-/chg. package, +/-/chg. class	component, package, class	Motive	Java	-	-	1.5k classes	-	-	-
Yoo and Choi [138]	Arch.	MDG	system	-	system	-	XML	-	-	-	-	-	-
de Boer et al. [139]	Arch.	DG	component, process, data object	+/-/chg. component, +/-/chg. process, +/-/chg. data object	component, process, data object	-	ArchiMate	-	-	-	-	-	-
Vora [140]	Arch.	ER, CG	class, method	-	component	-	TeCFRADL	-	-	10 kLOC	-	-	-
Feng and Maletic [141]	Arch.	ER, SL	component, interface, method	+/- interface, +/- method	component, interface, method	SOCIAT	UML	-	-	-	-	-	-
Tang et al. [142]	Arch.	PM	entire UML	-	entire UML	AREL	UML	-	-	-	-	-	-
Zhao et al. [143]	Arch.	SL	component, connector	-	component, connector	Ciasa	Wright	-	-	-	-	-	-
Wong and Cai [101]	Arch.	PM, HM	class	-	class	-	UML	-	-	14 revisions	0.021	0.061	-
van den Berg [144]	Arch.	TR, DG	adaptable	-	adaptable	no	UML	-	-	-	-	-	-
ten Hove et al. [149]	Req.	ER	requirement	+/- requirement	requirement	plug-in for BluePrint	SysML	-	-	-	-	-	-
Hewitt and Rilling [145]	Req.	DG	scenario, component	-	scenario, component	extended UCMNav2	UCM	-	-				
Lock and Kotonya [148]	Req.	TR, PM	requirement	-	requirement	ARChIVisT	-	-	-	-	-	-	-
Hassine et al. [150]	Req.	SL	entire UCM spec.	-	entire UCM spec.	CIA Tool	UCM	-	-	-	-	-	-
Goknil et al. [151]	Req.	ER	requirement, predicate, relation	+/- requirement, +/- predicate, +/-/typ. relation	requirement	no	-	-	-	-	-	-	-
Lee et al. [152]	Req.	TR	goal, use case	-	goal, use case	-	-	-	-	-	-	-	-
Spijkerman [153]	Req.	TR, ER	requirement, constraint, property, relation	+/- requirement, +/-/chg. property, +/-/val. constraint, +/-/typ. relation	requirement	-	-	-	-	-	-	-	-
Jönsson [154]	Req.	IR	requirement	-	requirement	SVDLIBC	-	-	-	400 requirements	0.171	0.177	-

O'Neal [156]	Req.	TR	requirement, misc. artifacts	-	requirement	-	-	-	-	120 artifacts, 1100 traces	-	-	-
O'Neal and Carver [155]	Req.	TR	requirement	+/chg. requirement	requirement	-	-	-	-	-	-	-	-
Antoniol et al. [157]	Misc. Art.	HM	file	CVS record	file	no	-	-	-	10k files, 3.7 mLOC	-	-	-
Beyer and Noack [158]	Misc. Art.	HM	file	CVS record	file	StatCVS, $cvs2cl^2$, CrocoPat	-	-	-	4 mLOC, 3.9k files		-	-
Askari and Holt [159]	Misc. Art.	PM, HM	file	CVS record	file	no	-	-	-	-	-	-	-
Sherriff and Williams [160]	Misc. Art.	HM	file	change record	file	Matlab	-	T: $O(n^2)$	-	12k files, 240k revisions	-	-	-
Jashki et al. [161]	Misc. Art.	HM	file	CVS record	file	Matlab	-	-	-	3k files, 31k revisions	-	-	-
Nadi et al. [162]	Misc. Art.	HM	-	-	-	DRACA	-	-	-	27k changes	0.885	0.698	-
Hammad et al. [166]	Arch., Code	ER	C++ class, C++ method	+/- C++ class, +/- C++ method	UML class	srcTrace	C++, UML	-	-	550 files, 200 changes	-	-	-
Sharafat and Tahvildari [167]	Arch., Code	PM	class, method, variable	+/- method, +/-/val. variable	class	-	Java, UML	-	-	58 classes	0.702	-	-
Sharafat and Tahvildari [168]	Arch., Code	PM	class, method, variable	-	class	-	Java, UML	-	-	58 classes	0.707	-	-
Kotonya and Hutchinson [170]	Arch., Req.	DG	component, requirement	+/- component, chg. property, chg. constraint	component, requirement	ECO-ADM	CADL	-	-	-	-	-	-
Xiao et al. [171]	Req., Code	CG, ER	BPEL task	+/- task, chg. task-property, chg. task-data	method	-	BPEL	-	-	-	-	-	-
Bohner [37], [172]	Arch., Code	DG	-	-	-	-	-	-	-	-	-	-	-
Bohner and Gracanin [173]	Arch., Code	DG	-	-	-	-	-	-	-	-	-	-	-
Hutchinson et al. [169]	Arch., Req.	TR	component, requirement	-	component, requirement	no	CADL	-	-	-	-	-	-
Khan and Lock [174]	Arch., Req.	TR	component, use case	-	component	-	-	-	-	-	-	-	-
Yu et al. [175]	Arch., Req.	CG	component, requirement	-	component	-	-	-	-	-	-	-	-
Briand et al. [176]	Arch., Req.	TR	class, method, sequence, use case, variable, message, test case	+/- use case, +/-/chg. message, +/-/sig./chg. method, +/-/vis./typ. variable	test case	RTSTool	UML	-	-	320k test cases	-	-	-

Ibrahim et al. [121]–[123]	Arch., Req., Code	TR	class, method, requirement, test case	chg. method	class, method, requirement, test case	Catia	C++, UML	-	-	4 kLOC	-	-	-
von Knethen and Grund [177]	Arch., Req.	TR, IR	entire UML	-	entire UML	QuaTrace	UML	-	-	-	-	-	-
Kim et al. [163]	Arch., Code	DG	source file, class, method, variable	-	source file, class, method, variable	iCIA	C, C++	-	-	7 mLOC	-	-	3 min
Hassan et al. [180]	Arch., Code	ER	component, interface, connector, port	+/- component, +/- interface, +/- connector, +/- port	component, interface, connector, port	set of Eclipse plug-ins	Ada, Perl, PHP, Java, AADL, XADL 2.0	-	-	-	-	-	-
Looman [179]	Arch., Req.	TR	component, requirement	+/-/chg. req., +/- req. predicate, +/-/chg. component	component, requirement	Alloy	AADL	-	-	-	-	-	-

TABLE II: All studies classified according to the criteria of our taxonomy [9]

REFERENCES

[1] V. Rajlich and P. Gosavi, "Incremental change in object-oriented programming," *IEEE Software*, vol. 21, no. 4, pp. 62–69, 2004.

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

[3] S. A. Bohner, "Impact analysis in the software change process: a year 2000 perspective," in *Proceedings of the 12th International Conference on Software Maintenance (ICSM'96)*, Monterey, CA, November 1996, pp. 42–51.

[4] 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.

[5] K. H. Bennett, "An introduction to software maintenance," *Information and Software Technology*, vol. 12, no. 4, pp. 257–264, 1990.

[6] M. Lee, A. J. Offutt, and R. T. Alexander, "Algorithmic analysis of the impacts of changes to object-oriented software," in *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 34)*, Santa Barbara, CA , USA, July 2000, pp. 61–70.

[7] R. S. Arnold and S. A. Bohner, "Impact analysis - towards a framework for comparison," in *Proceedings of the Conference on Software Maintenance (CSM '93)*, Montreal, Que., Canada, September 1993, pp. 292–301.

[8] S. A. Bohner, "A graph traceability approach for software change impact analysis," Ph.D. dissertation, George Mason University, Fairfax, VA, USA, 1995.

[9] 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.

[10] G. Tóth, P. Hegedűs, A. Beszédes, T. Gyimóthy, and J. Jász, "Comparison of different impact analysis methods and programmer's opinion: an empirical study," in *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10)*, New York, USA, 2010.

[11] M. Lindvall, "Evaluating impact analysis - a case study," *Empirical Software Engineering*, vol. 2, no. 2, pp. 152–158, 1997.

[12] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, Scotland, 2004, pp. 491–500.

[13] B. Breech, M. Tegtmeyer, and L. Pollock, "A comparison of online and dynamic impact analysis algorithms," in *Proceedings of the European Conference on Software Maintenance and Reengineering 2005*, March 2005, pp. 143–152.

[14] L. Hattori, G. d. Santos Jr., F. Cardoso, and M. Sampaio, "Mining software repositories for software change impact analysis: A case study," in *Proceedings of the 23rd Brazilian symposium on Databases*, Campinas, Sao Paulo, Brazil, October 2008, pp. 210–223.

[15] L. Hattori, D. Guerrero, J. Figueiredo, J. a. Brunet, and J. Damasio, "On the precision and accuracy of impact analysis techniques," in *Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*, Portland, OR, May 2008, pp. 513–518.

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

[17] R. Robbes, M. Lanza, and D. Pollet, "A benchmark for change prediction," Faculty of Informatics, Universit della Svizzerra Italiana, Lugano, Switzerland, Tech. Rep. 06, October 2008.

[18] H. Kagdi, M. L. Collard, and J. I. Maletic, "Towards a taxonomy of approaches for mining of source code repositories," in *Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR '05)*, New York, 2005, pp. 90–94.

[19] ——, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, pp. 77–131, 2007.

[20] H. Kagdi, "Mining software repositories to support software evolution," Ph.D. dissertation, Kent State University, August 2008.

[21] M. Kilpinen, "The emergence of change at the systems engineering and software design interface - an investigation of impact analysis," Ph.D. dissertation, Cambridge University, Engineering Department, 2008.

[22] B. J. Williams and J. C. Carver, "Characterizing software architecture changes: A systematic review," *Information and Software Technology*, vol. 52, no. 1, pp. 31–51, July 2009.

[23] ITU-T, "Recommendation ITU-T Z.151 User requirements notation (URN) – Language definition," ITU-T, Nov 2008.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip, "Chianti: A change impact analysis tool for Java programs," in *Proceedings of the 27th international conference on Software Engineering (ICSE '05)*. New York, NY, USA: ACM, 2005, pp. 664–665.

[29] X. Ren, "Change impact analysis for Java programs and applications," Ph.D. dissertation, New Brunswick Graduate School, Rutgers University, New Brunswick, New Jersey, USA, October 2007.

[30] F. Xia and P. Srikanth, "A change impact dependency measure for predicting the maintainability of source code," in *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC '04)*, vol. 2, September 2004, pp. 22–23.

[31] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: a control call graph based technique," in *Proceedings of the 12th Asia-Pacific Conference on Software Engineering (APSEC '05)*, December 2005, p. 9.

[32] L. C. Briand, J. Wuest, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, Oxford , UK, 1999, pp. 475–482.

[33] 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.

[34] V. Rajlich, "A model for change propagation based on graph rewriting," in *Proceedings of the 13th International Conference on Software Maintenance (ICSM '97)*, Bari, Italy, October 1997, pp. 84–91.

[35] G. Pirklbauer, C. Fasching, and W. Kurschl, "Improving change impact analysis with a tight integrated process and tool," in *Proceedings of the Seventh International Conference on Information Technology*, Las Vegas, Nevada, USA, April 2010, pp. 956–961.

[36] C. Fasching, "A tool for software visualization to support Impact Analysis (in German: Ein Visualisierungswerkzeug zur Unterstützung der Auswirkungsanalyse)," Master's thesis, Upper Austria University of Applied Sciences, Hagenberg, Austria, 2009.

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

[38] M. Zalewski and S. Schupp, "Change impact analysis for generic libraries," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, Pennsylvania, September 2006, pp. 35–44.

[39] M. Petrenko and V. Rajlich, “Variable granularity for improving precision of impact analysis,” in *Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC ’09)*, Vancouver, BC, May 2009, pp. 10–19.

[40] S. Black, “Computing ripple effect for software maintenance,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, pp. 263–279, 2001.

[41] S. S. Yau, J. S. Collofello, and T. M. McGregor, “Ripple effect analysis of software maintenance,” in *Proceedings Computer Software and Applications Conference (COMPSAC 78).* IEEE Computer Society Press: Piscataway NJ, 1978, pp. 60–65.

[42] H. Bilal and S. Black, “Computing ripple effect for object oriented software,” in *Proceedings of the 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE ’06)*, Nantes, France, July 2006, pp. 51–60.

[43] 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.

[44] 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 *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2007)*, Paris, October 2007, pp. 295–304.

[45] J. Jász, A. Beszédes, T. Gyimóthy, and V. Rajlich, “Static execute after/before as a replacement of traditional software dependencies,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM ’08)*, Beijing, October 2008, pp. 137–146.

[46] K. Chen and V. Rajlich, “RIPPLES: Tool for change in legacy software,” in *Proceedings of the IEEE International Conference on Software Maintenance*, Florence, Italy, November 2001, pp. 230–239.

[47] S. Gwizdala, Y. Jiang, and V. Rajlich, “Jtracker - a tool for change propagation in java,” in *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, March 2003, pp. 223–229.

[48] L. Bishop, “Incremental impact analysis for object-oriented software,” Master’s thesis, Iowa State University, 2004.

[49] F. Tip, “A survey of program slicing techniques,” *Journal of Programming Languages*, vol. 3, pp. 121–189, 1994.

[50] 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.

[51] K. B. Gallagher and J. R. Lyle, “Using program slicing in software maintenance,” *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751–761, August 1991.

[52] M. Hutchins and K. Gallagher, “Improving visual impact analysis,” in *Proceedings of the 14th IEEE International Conference on Software Maintenance (ICSM’98)*, Bethesda, Maryland, USA, March 1998, pp. 294–303.

[53] P. Tonella, “Using a concept lattice of decomposition slices for program understanding and impact analysis,” *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 495–509, June 2003.

[54] D. Binkley and M. Harman, “Locating dependence clusters and dependence pollution,” in *Proceedings of the 21st International Conference on Software Maintenance (ICSM’05)*, September 2005, pp. 177–186.

[55] 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.

[56] R. Santelices and M. J. Harrold, “Probabilistic slicing for predictive impact analysis,” Georgia Tech Center for Experimental Research in Computer Systems (CERCS), Tech. Rep., 2010.

[57] J. Law and G. Rothermel, “Whole program path-based dynamic impact analysis,” in *Proceedings of the International Conference on Software Engineering (2003)*, 2003, pp. 308–318.

[58] ——, “Incremental dynamic impact analysis for evolving software systems,” in *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE’03)*, November 2003, pp. 430–441.

[59] A. Orso, T. Apiwattanapong, and M. J. Harrold, “Leveraging field data for impact analysis and regression testing,” in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE’03)*, Helsinki, Finland, 2003, pp. 128–137.

[60] B. Breech, A. Danalis, S. Shindo, and L. Pollock, “Online impact analysis via dynamic compilation technology,” in *Proceedings of the 20th IEEE International Conference of Software Maintenance*, September 2004, pp. 453–457.

[61] B. Breech, M. Tegtmeyer, and L. Pollock, “Integrating influence mechanisms into impact analysis for increased precision,” in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM ’06)*, Philadelphia, PA, December 2006, pp. 55–65.

[62] T. Apiwattanapong, A. Orso, and M. J. Harrold, “Efficient and precise dynamic impact analysis using execute-after sequences,” in *Proceedings of the International Conference on Software Engineering (ICSE 2005)*, St. Louis, MO, May 2005, pp. 432–441.

[63] L. Huang and Y.-T. Song, “Dynamic impact analysis using execution profile tracing,” in *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications (SERA’06)*, Seattle, Washington, August 2006, pp. 237–244.

[64] A. Beszédes, T. Gergely, S. Faragó, T. Gyimóthy, and F. Fischer, “The dynamic function coupling metric and its use in software evolution,” in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR’07)*, Amsterdam, the Netherlands, March 2007, pp. 103–112.

[65] C. Gupta, Y. Singh, and D. S. Chauhan, “An efficient dynamic impact analysis using definition and usage information,” *International Journal of Digital Content Technology and its Applications*, vol. 3, no. 4, pp. 112–115, 2009.

[66] ——, “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.

[67] R. Vanciu and V. Rajlich, “Hidden dependencies in software systems,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM ’10)*, Timisoara, September 2010, pp. 1–10.

[68] J. Han, “Supporting impact analysis and change propagation in software engineering environments,” Monash University, Peninsula School of Computing & Information Technology, McMahons Road, Frankston, Victoria 3199, Australia, Tech. Rep. 96-09, October 1996.

[69] M. A. Chaumun, H. Kabaili, R. K. Keller, and F. Lustman, “A change impact model for changeability assessment in object-oriented software systems,” in *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, 1999, pp. 130–149.

[70] E. Arisholm, L. C. Briand, and A. Foyen, “Dynamic coupling measurement for object-oriented software,” *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 491–506, 2004.

[71] 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.

[72] D. Binkley and D. Lawrie, “Information retrieval applications in software maintenance and evolution,” in *Encyclopedia of Software Engineering*, P. Laplante, Ed. Taylor & Francis LLC, 2010, ch. 2.

[73] 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.

[74] S. Vaucher, H. Sahraoui, and J. Vaucher, “Discovering new change patterns in object-oriented systems,” in *Proceedings of the 2008 15th Working Conference on Reverse Engineering (WCRE ’08)*, Washington, DC, USA, 2008, pp. 37–41.

[75] 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.

[76] Y. Zhou, M. Wuersch, E. Giger, H. Gall, and J. Lue, “A bayesian network based approach for change coupling prediction,” in *Proceedings of the 15th Working Conference on Reverse Engineering 2008*, October 2008, pp. 27–36.

[77] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the probability of change in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 601–614, July 2005.

[78] M. Abdi, H. Lounis, and H. Sahraoui, "Analyzing change impact in object-oriented systems," in *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, Cavtat/Dubrovnik (Croatia), August 2006, pp. 310–319.

[79] ——, "Using coupling metrics for change impact analysis in object-oriented systems," in *Proceedings of the 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE '06)*, Nantes, France, July 2006, pp. 61–70.

[80] ——, "A probabilistic approach for change impact prediction in object-oriented systems," in *Proceedings of the 2nd Workshop on Artificial Intelligence Techniques in Software Engineering (AISEW 2009)*, Thessaloniki, Greece, April 2009, pp. 189–200.

[81] ——, "Predicting change impact in object-oriented applications with bayesian networks," in *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, Seattle, WA, July 2009, pp. 234–239.

[82] S. Mirarab, A. Hassouna, and L. Tahvildari, "Using bayesian belief networks to predict change propagation in software systems," in *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC '07)*, Banff, Alberta, Canada, June 2007, pp. 177–188.

[83] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, Timisoara, Romania, September 2010, pp. 1–10.

[84] B. Fluri, H. C. Gall, and M. Pinzger, "Fine-grained analysis of change couplings," in *Proceeding of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation 2005*, November 2005, pp. 66–74.

[85] 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.

[86] H. Gall, M. Jazayeri, and J. Krajewski, "CVS release history data for detecting logical couplings," in *Proceedings of the Sixth International Workshop on Principles of Software Evolution*, September 2003.

[87] 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.

[88] S. Ducasse, T. Gîrba, and J.-M. Favre, "Modeling software evolution by treating history as a first class entity," in *Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra '04)*, 2004, pp. 71–82.

[89] T. Gîrba and M. Lanza, "Visualizing and characterizing the evolution of class hierarchies," in *Proceeding of the Fifth International Workshop on Object-Oriented Reengineering (WOOR 2004)*, 2004.

[90] T. Gîrba, M. Lanza, and S. Ducasse, "Characterizing the evolution of class hierarchies," in *Proceeding of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, 2005, pp. 2–11.

[91] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, pp. 207–236, 2006.

[92] T. Gîrba, S. Ducasse, and M. Lanza, "Yesterdays Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes," in *Proceeding of the 20th IEEE International Conference on Software Maintenance (ICSM 04)*. IEEE Computer Society, 2004, pp. 40–49.

[93] T. Gîrba, S. Ducasse, and A. Kuhn, "Using concept analysis to detect co-change patterns," in *Proceedings of the Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, Dubrovnik, Croatia, 2007, pp. 83–89.

[94] 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.

[95] 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.

[96] S. Bouktif, Y.-G. Guéhéneuc, and G. Antoniol, "Extracting change-patterns from CVS repositories," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, Benevento, October 2006, pp. 221–230.

[97] T. Zimmermann and P. Weissgerber, "Preprocessing CVS data for fine-grained analysis," in *Proceedings of the 1st International Workshop on Mining Software Repositories*, 2004, pp. 2–6.

[98] R. Robbes and M. Lanza, "A change-based approach to software evolution," *Electronic Notes in Theoretical Computer Science*, vol. 166, pp. 93–109, January 2007.

[99] R. Robbes, M. Lanza, and M. Lungu, "An approach to software evolution based on semantic change," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, vol. 4422, pp. 27–41.

[100] R. Robbes and M. Lanza, "SpyWare: A change-aware development toolset," in *Proceedings of the 30th international conference on Software engineering (ICSE '08)*, Leipzig, Germany, May 2008, pp. 847–850.

[101] S. Wong and Y. Cai, "Predicting change impact from logical models," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '09)*, Edmonton, AB, Canada, September 2009, pp. 467–470.

[102] H. Kagdi and J. I. Maletic, "Software-change prediction: Estimated+actual," in *Proceedings of the Second International IEEE Workshop on Software Evolvability (SE '06)*, Philadelphia, PA, September 2006, pp. 38–43.

[103] H. Kagdi, "Improving change prediction with fine-grained source code mining," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, New York, NY, USA, 2007, pp. 559–562.

[104] H. Kagdi and J. I. Maletic, "Combining single-version and evolutionary dependencies for software-change prediction," in *Proceedings of 4th International Workshop on Mining Software Repositories (MSR'07)*, Minneapolis, MN, May 2007, pp. 107–110.

[105] D. M. German, A. E. Hassan, and G. Robles, "Change impact graphs: Determining the impact of prior code changes," *Information and Software Technology*, vol. 51, pp. 1394–1408, 2009.

[106] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, Como, Italy, September 2005, pp. 29–38.

[107] ——, "Fine grained indexing of software repositories to support impact analysis," in *Proceedings of the International Workshop on Mining Software Repositories (MSR'06)*, 2006, pp. 105–111.

[108] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta, "An eclectic approach for change impact analysis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, vol. 2, 2010, pp. 163–166.

[109] 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 *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, 2010.

[110] H. Kabaili, R. K. Keller, and F. Lustman, "A change impact model encompassing ripple effect and regression testing," in *Proceedings of the Fifth International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Budapest, Hungary, 2001, pp. 25–33.

[111] J.-P. Queille, J.-F. Voidrot, N. Wilde, and M. Munro, "The impact analysis task in software maintenance: A model and a case study," in *Proceedings of the International Conference on Software Maintenance*, Victoria, BC, Canada, September 1994, pp. 234–242.

[112] S. Barros, T. Bodhuin, A. Escudie, J. Queille, and J. Voidrot, "Supporting impact analysis: a semi-automated technique and associated tool," in *Proceedings of the 11th International Conference on Software Maintenance (ICSM'95)*, Opio (Nice), France, October 1995, pp. 42–51.

[113] L. Huang and Y.-T. Song, "Precise dynamic impact analysis with dependency analysis for object-oriented programs," in *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, Busan, August 2007, pp. 374–384.

[114] ——, "A dynamic impact analysis approach for object-oriented programs," in *Proceedings of the Conference on Advanced Software Engineering and Its Applications (ASEA '08)*, Hainan Island, December 2008, pp. 217–220.

[115] R. J. Walker, R. Holmes, I. Hedgeland, P. Kapur, and A. Smith, "A lightweight approach to technical risk estimation via probabilistic impact analysis," in *Proceedings of the 2006 international workshop on Mining software repositories (MSR '06)*, Shanghai, China, 2006, pp. 98–104.

[116] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, and D. D. S. Guerrero, "The hybrid technique for object-oriented software change impact analysis," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR '10)*, Madrid, Spain, March 2010, pp. 252–255.

[117] S. Wong, Y. Cai, and M. Dalton, "Change impact analysis with stochastic dependencies," Drexel University Philadelphia, PA, USA, Tech. Rep., 2011.

[118] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *Proceedings of the 17th IEEE Working Conference on Reverse Engineering (WCRE'10)*, Beverly, Massachusetts, USA, October 2010, pp. 119–128.

[119] X. Sun, B. Li, C. Tao, and S. Zhang, "HSM-based change impact analysis of object-oriented Java programs," *Chinese Journal of Electronics*, vol. 20, no. 2, pp. 247–251, April 2011.

[120] R. N. Mohamad, "A change impact analysis approach using visualization method," Master's thesis, Malaysia University of Technology, Faculty of Computer Science and Information Systems, 2010.

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

[122] ——, "A requirements traceability to support change impact analysis," *Asean Journal of Information Technology*, vol. 4, no. 4, pp. 345–355, 2005.

[123] ——, "A software traceability validation for change impact analysis of object oriented software," in *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006*, vol. 1, Las Vegas, Nevada, USA, June 2006, pp. 453–459.

[124] M. L. Lee, "Change impact analysis of object-oriented software," Ph.D. dissertation, Graduate Faculty of George Mason University, Fairfax, Virginia, 1998.

[125] D. Popescu, J. Garcia, K. Bierhoff, and N. Medvidovic, "Helios: Impact analysis for event-based systems," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, 2010.

[126] D. Popescu, "Impact analysis for event-based components and systems," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, vol. 2, New York, USA, 2010.

[127] L. Moonen, "Lightweight impact analysis using island grammars," in *Proceedings of the 10th International Workshop on Program Comprehension*, December 2002, pp. 219–228.

[128] M. A. Hoffman, "Automated impact analysis of object-oriented software systems," in *Proceedings of Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '03)*, Anaheim, CA, October 2003, pp. 72–73.

[129] A. Aryani, I. D. Peake, M. Hamilton, and H. Schmidt, "Change propagation analysis using domain information," in *Australian Software Engineering Conference 2009*, Gold Coast, Australia, April 2009, pp. 34–43.

[130] A. Aryani, I. D. Peake, and M. Hamilton, "Domain-based change propagation analysis: An enterprise system case study," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '10)*, Timisoara, September 2010, pp. 1–9.

[131] L. Briand, Y. Labiche, and L. O'Sullivan, "Impact analysis and change management of UML models," in *Proceedings of the 19th International Conference on Software Maintenance*, 2003, pp. 256–265.

[132] 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.

[133] C. Dantas, L. Murta, and C. Werner, "Mining change traces from versioned UML repositories," in *Proceedings of the Brazilian Symposium on Software Engineering (SBES'07)*, 2007, pp. 236–252.

[134] 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.

[135] ——, "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.

[136] ——, "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.

[137] A. McNair, D. M. German, and J. Weber-Jahnke, "Visualizing software architecture evolution using change-sets," in *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE '07)*, Vancouver, BC, October 2007, pp. 130–139.

[138] N. Yoo and H.-A. Choi, "An XML-based approach for interface impact analysis in sustained system," in *Proceedings of the International Conference on Information and Knowledge Engineering (IKE'04)*, Las Vegas, Nevada, USA, June 2004, pp. 161–167.

[139] F. de Boer, M. Bonsangue, L. Groenewegen, A. Stam, S. Stevens, and L. van der Torre, "Change impact analysis of enterprise architectures," in *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI)*, August 2005, pp. 177–181.

[140] U. Vora, "Change impact analysis and software evolution specification for continually evolving systems," in *Proceedings of the Fifth International Conference on Software Engineering Advances*, Nice, France, August 2010, pp. 238–243.

[141] 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.

[142] A. Tang, A. Nicholson, Y. Jin, and J. Han, "Using bayesian belief networks for change impact analysis in architecture design," *The Journal of Systems and Software*, vol. 80, pp. 127–148, 2007.

[143] J. Zhao, H. Yang, L. Xiang, and B. Xu, "Change impact analysis to support architectural evolution," *Journal of Software Maintenance*, vol. 14, no. 5, pp. 317–333, 2002.

[144] K. van den Berg, "Change impact analysis of crosscutting in software architectural design," in *Proceedings of the Workshop on Architecture-Centric Evolution (ACE 2006)*, Nantes, July 2006, pp. 1–15.

[145] J. Hewitt and J. Rilling, "A light-weight proactive software change impact analysis using use case maps," in *Proceedings of the IEEE International Workshop on Software Evolvability (Software-Evolvability'05)*, Budapest, Hungary, September 2005, pp. 41–48.

[146] N. Nurmuliani, D. Zowghi, and S. P. Williams, "Requirements volatility and its impact on change effort: Evidence-based research in software development projects," in *Proceedings of the 11th Australian Workshop on Requirements Engineering*, Adelaide, Australia, 2006.

[147] N. Mellegård and M. Staron, "Improving efficiency of change impact assessment using graphical requirement specifications: An experiment," in *Product-Focused Software Process Improvement*. Springer Berlin / Heidelberg, 2010, vol. 6156, pp. 336–350.

[148] S. Lock and G. Kotonya, "An integrated, probabilistic framework for requirement change impact analysis," *Australasian Journal of Information Systems*, vol. 6, no. 2, pp. 38–63, September 1999.

[149] D. ten Hove, A. Goknil, I. Kurtev, K. Berg van den, and K. Goede de, "Change impact analysis for sysml requirements models based on semantics of trace relations," in *Proceedings of the ECMDA Traceability Workshop (ECMDA-TW)*, Enschede, the Netherlands, June 2009, pp. 17–28.

[150] J. Hassine, J. Rilling, J. Hewitt, and R. Dssouli, "Change impact analysis for requirement evolution using use case maps," in

Proceedings of the 8th International Workshop on Principles of Software Evolution, 2005, pp. 81–90.

[151] A. Goknil, I. Kurtev, and K. van den Berg, "Change impact analysis based on formalization of trace relations for requirements," in *Proceedings of the EC-MDA Traceability Workshop (ECMDA-TW)*, 2008, pp. 59–75.

[152] W.-T. Lee, W.-Y. Deng, J. Lee, and S.-J. Lee, "Change impact analysis with a goal-driven traceability-based approach," *International Journal of Intelligent Systems*, vol. 25, pp. 878–908, August 2010.

[153] W. Spijkerman, "Tool support for change impact analysis in requirement models - exploiting semantics of requirement relations as traceability relations," Master's thesis, Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, October 2010.

[154] P. Jönsson, "Impact analysis organisational views and support techniques," Ph.D. dissertation, Department of Systems and Software Engineering School of Engineering Blekinge Institute of Technology Sweden, 2005.

[155] J. S. O'Neal and D. L. Carver, "Analyzing the impact of changing requirements," in *Proceedings of the IEEE International Conference on Software Maintenance*, Florence, Italy, November 2001, pp. 190–195.

[156] J. S. O'Neal, "Analyzing the impact of changing software requirements: A traceability-based methodology," Ph.D. dissertation, Louisiana State University, 2003.

[157] G. Antoniol, V. F. Rollo, and G. Venturi, "Detecting groups of co-changing files in CVS repositories," in *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, Lisbon, Portugal, September 2005, pp. 23–32.

[158] D. Beyer and A. Noack, "Clustering software artifacts based on frequent common changes," in *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, 2005, pp. 259–268.

[159] M. Askari and R. Holt, "Information theoretic evaluation of change prediction models for large-scale software," in *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR06)*, New York, 2006, pp. 126–132.

[160] M. Sherriff and L. Williams, "Empirical software change impact analysis using singular value decomposition," IBM, North Carolina State University, Tech. Rep., 2007.

[161] M.-A. Jashki, R. Zafarani, and E. Bagheri, "Towards a more efficient static software change impact analysis method," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '08)*, 2008.

[162] S. Nadi, R. Holt, and S. Mankovskii, "Does the past say it all? using history to predict change sets in a CMDB," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, Madrid, Spain, March 2010, pp. 97–106.

[163] T.-h. Kim, K. Kim, and W. Kim, "An interactive change impact analysis based on an architectural reflexion model approach," in *Proceedings of the IEEE 34th Annual Computer Software and Applications Conference (COMPSAC '10)*, Seoul, July 2010, pp. 297–302.

[164] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap betwen source and high-level models," in *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT '95)*, Washington, D.C., USA, October 1995, pp. 18–28.

[165] ——, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, April 2001.

[166] M. Hammad, M. L. Collard, and J. I. Maletic, "Automatically identifying changes that impact code-to-design traceability," in *Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC '09)*, Vancouver, BC, May 2009, pp. 20–29.

[167] A. R. Sharafat and L. Tahvildari, "A probabilistic approach to predict changes in object-oriented software systems," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR '07)*, Amsterdam, Netherlands, March 2007, pp. 27–38.

[168] ——, "Change prediction in object-oriented software systems: A probabilistic approach," *Journal of Software*, vol. 3, no. 5, pp. 26–39, May 2008.

[169] J. Hutchinson, G. Kotonya, B. Bloin, and P. Sawyer, "Understanding the impact of change in COTS-based systems," in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '03)*, Las Vegas, USA, June 2003.

[170] G. Kotonya and J. Hutchinson, "Analysing the impact of change in COTS-based systems," *Lecture Notes in Computer Science*, vol. 3412, pp. 212–222, 2005.

[171] H. Xiao, J. Guo, and Y. Zou, "Supporting change impact analysis for service oriented business applications," in *Proceedings of the International Workshop on Systems Development in SOA Environments (SDSOA '07)*, 2007, pp. 6–11.

[172] S. A. Bohner, "Software change impacts-an evolving perspective," in *Proceedings of the International Conference on Software Maintenance*, 2002, pp. 263–272.

[173] S. A. Bohner and D. Gracanin, "Software impact analysis in a virtual environment," in *Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop*, December 2003, pp. 143–151.

[174] S. S. Khan and S. Lock, "Concern tracing and change impact analysis: An exploratory study," in *Proceedings of the 2009 ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design*, Vancouver, BC, Canada, May 2009, pp. 44–48.

[175] B. Yu, A. Mili, W. Abdelmoez, R. Gunnalan, M. Shereshevsky, and H. H. Ammar, "Requirements change impact in software architecture."

[176] 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.

[177] A. von Knethen and M. Grund, "QuaTrace: a tool environment for (semi-) automatic impact analysis based on traces," in *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, September 2003, pp. 246–255.

[178] 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, January 1998.

[179] S. Looman, "Impact analysis of changes in functional requirements in the behavioral view of software architectures," Master's thesis, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, 2009.

[180] M. O. Hassan, L. Deruelle, and H. Basson, "A knowledge-based system for change impact analysis on software architecture," in *Proceedings of the Fourth International Conference on Research Challenges in Information Science (RCIS)*, Nice, France, May 2010, pp. 545–556.

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

[182] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "JRipples: A tool for program comprehension during incremental change," in *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 05)*, May 2005, pp. 149–151.