Mining Cause-Effect-Chains from Version Histories

Kim Herzig
Software Engineering Chair
Saarland University, Germany
Email: herzig@cs.uni-saarland.de

Andreas Zeller
Software Engineering Chair
Saarland University, Germany
Email: zeller@cs.uni-saarland.de

***Abstract*—Software reliability is determined by software changes. How do these changes relate to each other? By analyzing the impacted method definitions and usages, we determine *dependencies* between changes, resulting in a *change genealogy* that captures how earlier changes enable and cause later ones. Model checking this genealogy reveals *temporal process patterns* that encode key features of the software process: "Whenever class A is changed, its test case is later updated as well." Such patterns can be validated automatically: In an evaluation of four open source histories, our prototype would recommend pending activities with a precision of 60–72%.**

***Keywords*-Distribution; Maintenance; Enhancement; Life cycle; Software process models**

I. Introduction

Software and its reliability is a product of its history, which can be characterized as a sequence of changes. By mining recorded changes, one can identify frequently changing components—an important factor in predicting the risk of defects. And one can discover sets of components that changed frequently together, revealing couplings that are inaccessible to program analysis.

Research studies in analyzing software history have been mostly constrained to either *space* or *time*. Being constrained to *space* means that one examines the evolution of single components, aggregating features over time. Being constrained to *time* means that one examines which components were changed at a single moment in time, extracting co-changes from the resulting transactions. What we would like to have is reasoning over *multiple* components at *multiple* points in time. However, such reasoning requires a holistic view of *all* changes to *all* components.

In this paper, we present and evaluate a new concept that enables such multi-dimensional reasoning. A *change genealogy* orders changes by *dependencies*; it tells how changes influence and cause each other. By mining and *model checking* such genealogies, we obtain frequent *temporal patterns* that encode key features of the software process that span both space and time: "Whenever class A is changed, its test case is later updated as well." or "Whenever Tom changes his code, Anna has to respond on Tom's change". Once mined, such patterns can be used as building blocks for a software process model; they also can be validated and enforced automatically in further development.

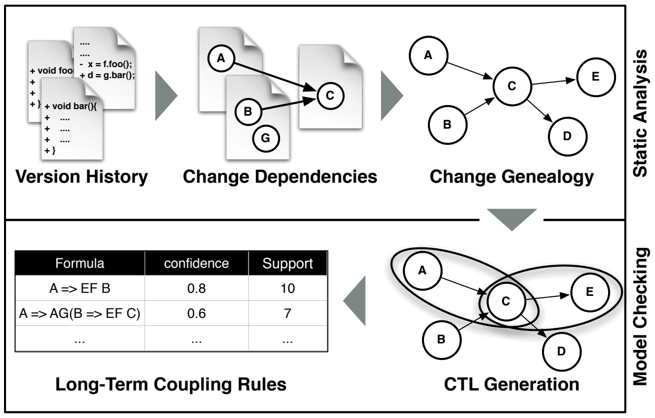

Figure 1. How GENEVA works. GENEVA takes a *version history* and determines *dependencies* between changes. These dependencies form a graph, the *change genealogy. Model checking* reveals *frequent rules* that describe temporal key features of the underlying software process.

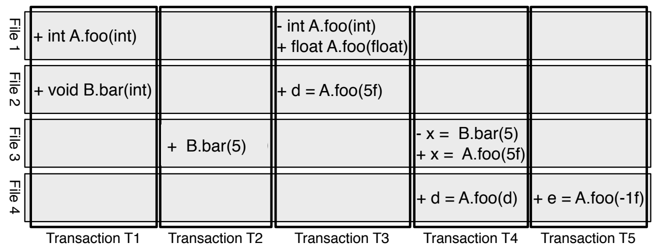

Figure 2. We characterize code changes by the method calls and definitions they add or change. Changes depend on each other based on the affected methods.

Let us illustrate our approach in a nutshell (Figure 1). The key idea is to establish *dependencies* between changes—a change T_2 *depends* upon a change T_1 if T_2 uses or modifies code defined or previously changed within T_1. We find such dependencies by analyzing changes applied to method definitions and method usages. Figure 2 shows five changes modifying four files in two packages. T_1 adds two method definitions that are later used and changed by T_2 and T_3, respectively. T_2 and T_3 thus directly depend on T_1. Note that we do not rely on the assumption that transactions should leave the project in a compilable state. In code repositories that span multiple projects and in which code is shared among projects (e.g. Google) it is frequently the case that code changes leave a dependent project uncompilable.

Summarizing, all direct dependencies between changes

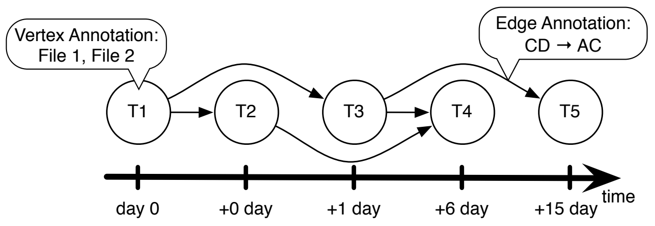

Figure 3. Sample change genealogy derived from the method operations shown in Figure 2. An arrow $T_i \rightarrow T_j$ indicates that the change T_j depends on T_i. Edges annotated contain the dependency types justifying the edge. Vertices annotations reference the changed file names.

result in a *change genealogy*—a dependency graph between changes: Figure 3 shows both the direct and transitive descendants of T_1. As change genealogies grow to thousands of changes, we can use them to *mine* for frequent patterns—in particular *temporal* patterns that span space and time. For this purpose, we associate each change with *features* such as the file or package being modified. We then use *model checking* to determine frequent rules. These rules are expressed in computation tree logic (CTL); a rule such as "$\mathit{file}_1 \Rightarrow \mathsf{EF}\ \mathit{test}_1$" means that whenever file_1 is changed, eventually, test_1 is changed as well. Additionally, the rule implies that the change to test_1 is structural dependent on the change applied to file_1. Such *long-term coupling* based on change dependencies is not detected by current mining approaches due to their restriction in space and time.

Mined temporal rules can become part of the formal software process model; generally speaking, they encode actions that typically follow immediately or after some period of time. Of course, they can also be validated and enforced automatically: Should file_1 be changed without test_1 following suit before release, chances are that file_1 is not properly tested. When file_1 is changed, we can thus give a *recommendation* reminding developers that test_1 must eventually be examined—or changed—as well. These recommendations are quite accurate: In an evaluation of four open source histories, our GENEVA[1] prototype would recommend pending activities with a precision of 60–72%.

The remainder of this paper is organized as follows. We first give an overview of the related work (Section II) before we define dependencies between changes (Section III) used to extract change genealogies (Section IV). Next, we describe how to extract temporal rules (Section V) and how to mine these rules (Section VI). We then evaluate the accuracy and usefulness of the mined rules both qualitatively (Section VII) and quantitatively (Section VIII). Section IX closes with conclusion and consequences.

II. Background

The contributions of this paper combine multiple common fields in mining software repositories. We present change dependency graphs as a model to analyze change impact. As usage example we present an approach to detect change couplings. Additionally, we use temporal logic to model development processes to provide recommendations for developers. To put our work in perspective, this section provides background information about previous work.

A. Change Dependencies

Stoerzer et al. [1] used a change classification technique and dependency definition very similar to the one used in this paper to detect dependencies between *atomic changes*. The main goal of their tools was to detect failure-inducing changes. Later, this work was extended by Wloka et al. [2] to identify committable code changes that can be applied to the version archive without causing existing tests to fail. Both approaches aim to detect change dependencies within one revision but not across long periods of time.

Brudaru and Zeller [3] were among the first presenting the concept of *change genealogies* that model the dependencies between atomic code changes spanning the complete software project lifetime. Later, German et al. [4] introduced the related concept of *change impact graphs* to identify those code changes that influenced the reported location of a failure. In contrast to our approach, German et al. [4] resolved change dependencies backwards for a given starting point in the source code to identify possible causes for a bug or failure. In this paper, we examine all change dependencies globally to detect and predict future cause-effect chains.

B. Change Impact Analysis

There are many approaches measuring the impact of single code changes dynamically. Ren et al. [5] decomposed code changes into sets of atomic changes to determine test cases whose execution behavior may have changed. Later, Stoerzer et al. [1] extended their work to successfully classify changes as fix inducing. Techniques like CoverageImpact [6] and PathImpact [7] measure the impact of atomic code changes on program execution.

But using dynamic techniques like program or test execution limit the impact analysis in two ways. First, it requires the software project to be in a compilable and executable state — which is not always the case during development. Second, it only reflects the immediate impact of code changes on program structure and execution behavior.

Alam et al. [8] used change dependency graphs [4] to examine how changes build on each other over time. The authors showed that dependencies between changes vary across different projects and that changes build on top of new code—instead on old stable code—are more defect prone.

In his PhD proposal [9], Herzig introduced the concept of *transaction dependency graphs* based on the notion of *change genealogies* defined by Brudaru and Zeller [3]. Similar to the *change impact graphs* by German et al. [4], Herzig defined dependency graphs but based on version control transactions instead of atomic changes to define multiple dependency metrics on these genealogy graphs.

[1]GENEVA = GENealogy Extraction from Version Archives.

C. Process Models and Temporal Logic

Cook and Wolf [10] were among the first to explore methods for automatic process modeling. Since then, a lot of approaches [11]–[13] model and analyze processes, workflows and execution traces. Instead of describing the behavioral aspects of a process, we model the relationship between code changes applied during development.

To extract CTL formulas, we generate formulas similar than Wasylkowski and Zeller [14] did. We use so-called temporal-logic queries [15] using multiple placeholders [16]. In their paper, Wasylkowski and Zeller [14] give a good and short introduction on CTL and Kirpke structures (Figure 4). In the remainder of the paper, we assume a basic knowledge of CTL and model checking using Kripke structures.

D. Change Couplings

In 1998, Gall et al. [17] were the first to examine CVS release data sets to detect *logical couplings* between modules. Later, Zimmermann et al. [18] and Ying [19] used association rule mining techniques to detect coupled changes, and to automatically suggest and predict likely further changes. Fluri et al. [20] showed that many change couplings are not caused by source code changes—and thus would be undetectable by program analysis.

Recently, Canfora et al. [21] used multivariate time series to detect cross-transaction change couplings. Using sliding windows the authors determine change couplings occurring within certain timeframes. In contrast to our approach, their approach does not consider any structural information and thus cannot distinct between change couplings caused by structural dependencies or by independent development activities frequently occurring short after each other.

III. Change Dependencies

How can we tell that one change depends on another one? Regular *version archives* contain sequences of code changes committed to the software project in temporal order. However, version archives do not explicitly state how these code changes depend on each other—because this information is rarely needed in regular development. The fact that a revision was committed after a previous one does not imply any dependency. To determine cause-effect chains, we must detect such inter-change dependencies—that is, determine which changes are based on each other. Deciding whether two changes T_i and T_j depend on each other is a hard problem. At the semantic level, it is generally undecidable to determine the effect of a change. At the syntactic level, one needs a full-fledged static analysis to determine whether applying T_j is possible without applying T_i first; in practice, this means an attempt to compile the program for each pair of changes. At the lexical level, one can determine whether T_i and T_j overlap each other in the affected locations; this is how version control systems detect conflicts between changes.

Table I
Change Categories: Code changes belong to none, one or multiple of these categories. These change operations are used to define dependency rules.

Category	Description
AD	Adding a new method definition
CD	Changing definition of method
RD	Removing definition of method
AC	Adding new method call
CC	Changing method call
RC	Removing method call

With the semantic level infeasible, the syntactic level too expensive, and the lexical level too primitive, we needed a compromise that would be scalable, yet precise. Similar to Ren et al. [5], we check which *methods* are affected by a change: If T_i introduces a method that T_j calls, then T_j depends on T_i. Code changes will be reduced to those *atomic changes* that change, add or remove method definitions and method calls. Depending on its change operation, each atomic change then gets categorized using the corresponding change categories shown in Table I. With this simple notion of code changes we can define a fixed set of *dependency rules* that describe possible change dependencies:

- Each change to a *method definition* depends on the previous definition of the very same method.
- Each change to a *method call* depends on the previous change to the very same method call and the change defining the current version of the called method.

Atomic changes categorized as CD or RD depend on those changes that previously added or changed the changed method definition. AC, CC and RC changes depend on the AD or CD of the definition of the called method. Additionally, CC and RC changes depend on those changes that added the method call before. AD changes depend on earlier but already deleted definitions of a method within the same class having the same method signature.

To find these change dependencies, we checkout all revisions from the version archive and detect method changes using abstract syntax trees derived from the partial program analysis tool by Dagenais and Hendren [22].

IV. Change Genealogies

On of our primary targets was to design a dependency model that does not rely on the assumption that a project is compilable at all revisions. In large code repositories that span multiple separate but dependent projects, this assumption gets frequently violated. A prominent example is the source code repository used within *Google Inc.*

We use the concept of *change genealogies* introduced by Brudaru and Zeller [3]—a directed, acyclic dependency graph that expresses dependencies between changes committed to the version repository over time and space. Each vertex in a change genealogy represents a *code change* committed to the version repository. Edges between two vertices express a *dependency* between both corresponding code changes. A dependency edge is directed and points from the initial change to the depending one. While most mining approaches treat version control systems as plain temporal sequence of code changes, genealogy graphs combine temporal and structural information about code changes spanning a dependency graph. Similar models have been used to analyze the impact of changes on the quality [8] or the propagation of changes through the system [4].

Since the concept of change genealogies is mostly theoretical, we had to define and modify some aspects of the original change genealogy definition to make genealogies applicable in practice. To fit dependencies between atomic changes (see Section III) into the concept of a change genealogy, we aggregate atomic changes applied simultaneously to the version archive into sets of atomic changes—so called *transactions*. In Figure 3, we show a transaction-based change genealogy graph corresponding to the artificial revision history shown in Figure 2. Each vertex corresponds to a transaction applied to the archive and is labeled by its unique identifier. Each vertex is annotated with the names of the source files that were changed within the transaction. Thus, change genealogies described within this paper model dependencies between transactions (time) and dependencies between changed source code files (space). All edges are directed and point from earlier changes to later changes—the future cannot influence the past. Edges between two vertices represent a direct structural dependency between the corresponding transactions. Each edge is labeled with the set of dependency rules that caused the edge to be added to the genealogy. The time gap an edge is bridging can be arbitrary.

Combining atomic changes and modeling dependencies between transactions disregards dependencies between changes applied together. The assumption is that changes that were applied together by the same author are interdependent and do not refer to long-term cause-effect chains. Furthermore, considering dependencies between changes applied in the same transaction would add many cycles to change genealogies and would therefore heavily affect our model checking approach (see Section V-C).

Our change genealogies are designed to model cross transaction dependencies using structural code dependencies that cannot be detected by standard mining techniques [18], [19], [21]. In general, change genealogies are not bound to the transaction level of granularity. But for the purpose of detecting long-term couplings this simplification reduces the amount of entities to be analyzed dramatically.

V. Long-Term Couplings

To demonstrate the expressive power of change genealogies and to give a practical working example how to use such change genealogy graphs, we implemented a tool that predicts long-term cause effect chains based on genealogy graphs. We build on the concept of *change couplings*, introduced by Gall et al. [17] and later improved by Zimmermann et al. [18] and Canfora et al. [21]: If two artifacts are coupled by frequent common changes, we can use this coupling to predict related changes. Such couplings are usually undetectable by program analysis [20].

We consider artifacts frequently or exclusively changed together (within the same transaction) to be strongly coupled. Using change couplings, it is possible to detect incomplete code changes and to give recommendations for further changes. The concept of change couplings has an important deficiency, though: It relies on the assumption that coupled artifacts get changed frequently (or always) together within the same transaction or at least in small, static time windows after each other. Often, artifacts that are frequently changed across multiple transactions by different authors get disregarded. As an example, consider a `login` function defined in project P_1 and a `service` class defined in project P_2. Changing the `login` function by throwing a new runtime exception requires the developer of class `service` to respond. Both files are maintained by different developers and thus would never occur within one transaction. Further, the dependency would not be detected by simply compiling all projects. Still, these files should be considered as tightly coupled. Tools like eROSE [18] do not detect these couplings because they analyze each transaction independently. Additionally, approaches like Canfora et al. [21] disregard structural information. Files changed frequently short after each other might not be structurally dependent but just got changed due to an iterative development process. To detect cross-transaction change couplings, we need structural dependency information between code changes. Considering the temporal order of transactions only does not suffice.

A. CTL on Genealogies

Change genealogies model dependencies between changes and can be used to extend the concept of change couplings. *Long-term change couplings* are change couplings spanning across multiple software revisions. In terms of change genealogies, we search for software artifact a_1 and a_2 such that whenever a_1 got changed, there exists a genealogy path from a_1 to a_2. The existence of such a path would imply that both changes structurally depend on each other. Each long-term coupling can be seen as a *cause-effect chain:* a developer changes her code and other developers respond to her change; the responding change can again cause other developers to respond to it.

To express long-term couplings, we use *computational tree logic*. CTL is the natural temporal logic interpreted over branching time structures introducing path qualifiers. The above example could be expressed as: $a_1 \Rightarrow \mathsf{EF}\, a_2$—read as "$a_1$ imply exists finally a_2", and means that on every path starting at a_1, there is at least one path going through a_2.

Using CTL to express long-term change couplings allows us to use *formal verification techniques* to *model check* long-term change couplings on change genealogies. While being able to express complex long-term dependency relations in logical formulas, we can also verify those relations automatically on any change genealogy at any time.

CTL is a powerful language with which you can express very complicated temporal properties on a change genealogy graph. We omit an introduction to CTL due to space reasons. Our goal is to find many plausible CTL formulas that might express long-term cause effect chains. To do so, we use *predefined CTL templates* [16] that are suitable to express long-term cause effect chains. Our CTL formulas make use of only three CTL operators: EF (exists finally), EX (exists next), and AG (all globally). We also limit the number of involved source code artifacts to three:

$Templ_1$: $a_1 \Rightarrow \mathsf{EF}\, a_2$: Changing a_1 causes at least one dependent change on a_2.

$Templ_2$: $a_1 \Rightarrow \mathsf{AG}(a_2 \Rightarrow \mathsf{EF}\, a_3)$: Changing a_1 and later changing a_2 causes a change in a_3. All later changes depend on the initial change of a_1.

$Templ_3$: $a_1 \Rightarrow \mathsf{EF}(a_2 \wedge a_3)$: Changing the artifact a_1 causes dependent changes in a_2 and a_3 within the same transaction.

$Templ_4$: $a_1 \Rightarrow \mathsf{EF}\, a_2 \wedge \mathsf{EF}\, a_3$: Changing a_1 causes dependent changes in a_2 and a_3 but not necessarily in the same transaction.

We do not claim that this set of CTL formulas is complete. In fact, you can choose any CTL formula that will match your purposes.

While CTL formulas are defined on software artifacts (e.g. source code files) a change genealogy graph expresses dependencies between sets of atomic changes. To bridge this gap, we can use the change genealogy transaction annotations. Replacing the artifact placeholders a_1,a_2, and a_3 within the CTL templates with the corresponding changed files, we can express temporal dependencies over change genealogy paths.

To illustrate this procedure, lets consider the change genealogy from Figure 3 using CTL template $Templ_1$. We choose a path that matches the temporal logic of $Templ_1$: $\{T_1, T_3, T_5\}$. Within $Templ_1$ we replace the variable a_1 with the file names changed within T_1: F_1, F_2 and the variable a_2 with those file names changed within T_5 : F_4. The resulting formulas are: $F_1 \Rightarrow \mathsf{EF}\, F_4$ and $F_2 \Rightarrow \mathsf{EF}\, F_4$. Later (Section V-C) we will see that not all possible template instances have to be generated.

Long-term couplings detected by GENEVA imply structural dependencies between coupled artifacts.

B. Limiting the Temporal Scope

In the previous section, we discussed how to generate CTL formulas using CTL templates. But in most projects all such dependencies will become eventually true. Even though each temporal path expressed by a long-term coupling rule is based on structural code dependencies, we want to limit the

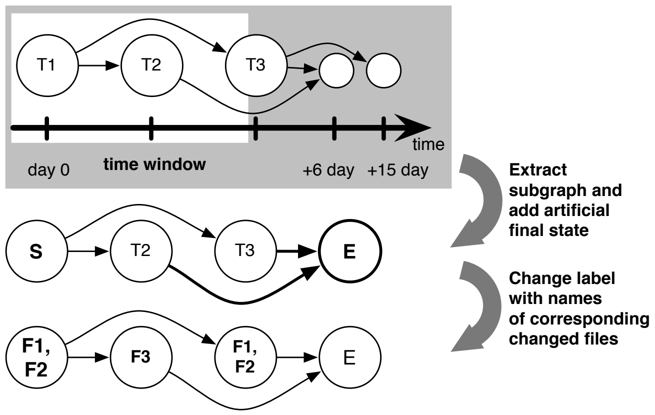
T1
T2
T3
time
day 0
time window
+6 day
+15 day
Extract subgraph and add artificial final state
S
T2
T3
E
Change label with names of corresponding changed files
F1, F2
F3
F1, F2
E

Figure 4. Extracting genealogy subgraphs from change genealogy using sliding time window and converting genealogy subgraph to Kripke structure.

temporal scope in which such long-term coupling rules must be valid. The longer the time between two dependent nodes, the lower the probability that the later applied change was caused by the earlier one. Therefore, we limit the number of days between the initial change and the depending changes. Doing so, we can ensure that there is a maximal time window (*max_days*) in which a CTL formula must be valid.

For this purpose, we use a sliding window approach to generate multiple *genealogy subgraphs* from the original genealogy graph. For each vertex u of the original genealogy graph $G(V, E)$ we generate a corresponding genealogy subgraph $G'(V', E')$ such that:

$V' = \{v \in V \mid t(u, v) \leq \mathit{max\_days} \ \wedge \ \mathit{path}(u, v)\} \cup \{u\}$ where $t(u, v)$ is the number of days between the commit dates of u and v. $\mathit{path}(u, v)$ is true if and only if there exists a path from u to v in G.

$E' = \{e(v_1, v_2) \in E \mid v_1 \in V' \ \wedge \ v_2 \in V'\}$

Figure 4 shows an example genealogy subgraph extracted from our initial genealogy in Figure 3 ($\mathit{max\_days} = 1$). Each genealogy subgraph is a connected graph having u (here T_1) as a root. The number of *generated* genealogy subgraphs equals the number of vertices in the original genealogy graph G. Graphs with a depth of one can be ignored.

Model checking the CTL formulas on the genealogy subgraphs ensures a time window in which these formulas have to be valid. Additionally, we automatically ignore changes with no outgoing dependency. This reduces the model checking space and makes model checking all these formulas feasible.

C. Model Checking Genealogies

At this point, we have introduced the concept of change genealogies and explained the concept of long-term change couplings described as CTL formulas. To *derive* valid long-term couplings, we have to *model check* which CTL formulas evaluate to true on the genealogy subgraphs. But before, we have to transform each genealogy subgraph into a *Kripke structure*—a nondeterministic finite state machine whose nodes represent reachable states with transition edges between them.

Our genealogy subgraphs from Section V-B are connected, directed, acyclic graphs and can be interpreted as a nondeterministic finite state machine. They therefore already are Kripke structures. It remains to add a new artificial final state (the graph will be left-total) and replacing the original node labels by those filenames changed by the corresponding transaction (using the vertex annotations). This way, our Kripke structures express temporal dependencies between changed files instead of transactions.

The resulting Kripke structures can be used to model check our CTL formulas. CTL formulas evaluating to true on at least one Kripke structure are worth further investigation since they represent potential long-term coupling rules.

VI. Long-Term Coupling Rules

In the previous section, we generated valid CTL formulas and prepared our data model to allow automatic CTL validation. But which of these formulas describe *frequent change patterns* and which formulas occurred rarely or only once? To mark important rules and to determine the strength of a coupling rule we rank rules by their *support* and *confidence* measures [18]:

Support. We measure the number of Kripke structures on which the CTL formula F was evaluated to be true as **support**(F). In our initial example (Figure 3), the formula $\mathit{File}_1 \Rightarrow \mathsf{EF}\,\mathit{File}_3$ has a support of two. File_1 was changed in transactions T_1 and T_3. The transactions T_2 and T_4 change File_3 and depend on either T_1 or T_3.

Confidence. To measure the strength of the consequence expressed in formula F we calculate **confidence**(F) as the fraction of the formula's support divided by the number of times the premises (source code file a_1) has been changed. In our initial example (Figure 3), the formula $\mathit{File}_1 \Rightarrow \mathsf{EF}\,\mathit{File}_3$ has a confidence value of one. The support value of the formula is two (see above) and File_1 got changed twice.

Rules are primarily ranked by confidence: The higher the confidence, the higher the rank of the rule. Rules with equal confidence are ranked by support.

A. Rules as Recommendations

The purpose of change couplings is to be used as recommendations. Whenever a programmer commits a change, a recommendation tool suggests further changes based on rules extracted from earlier code changes. Long-term change rules express frequent change rules that span multiple change transactions. Thus, each recommendation based on long-term change rules does not suggest code changes to be made within the same transaction but might indicate future development activities; further changes to be applied within a time window of *max_days* number of days.

The computation of long-term coupling rules is already described in Section V. To compute recommendations for a given transaction T_i, we have to execute the following steps:

1) Generate the change genealogy graph until transaction T_{i-1} and generate all valid CTL formulas within the current genealogy subgraph. That is, extract all changed files from all transactions of the genealogy subgraph and create all possible CTL formulas if not in the CTL cache. Each CTL formula is stored as *long-term coupling rule* together with the up-to-date support and confidence values.
2) Select all long-term coupling rules with implication premises that correspond to files changed within the transaction T_i.
3) Rank the selected rules by confidence and support.

The computation of change genealogies and CTL rules, model checking them and computing support and confidence values on the fly takes time. Generating all long-term coupling rules for a mid-size project spanning a history of 15,000 transactions take several hours—depending on the average number of files changed by transactions. (Of course, change genealogies do not have to be regenerated for each transaction but can be extended by single or multiple transactions.) Still, to improve efficiency, we used the following optimizations:

Ignoring large transactions. Transactions that touch many files are suspicious because most of them either combine multiple changes that should be separated or they refer to refactoring or documentation updates. In general, these transactions have a large number of incoming and outgoing dependency edges causing millions of CTL formulas generated. Additionally, long-term coupling rules derived from large transactions have a higher chance of causing many false positives.

We determine the median over the number of files changed by earlier transactions. Transactions for which the number of changed files is larger than the 3/4-quantil of the change size distribution are ignored.

Ignoring rarely changed files. In a project history, there are many code artifacts that have a very limited life span. Other artifacts are rarely updated, if ever. To minimize the number of relevant CTL formulas and to ease the memory consumption during model checking, we ignored all files that were changed only once.

Ignoring deleted files. When generating CTL formulas, we drop rules that would contain files deleted within the transaction as implication. Whenever a source code artifact gets deleted, we remove all CTL formulas that have the deleted artifacts as implication premise.

Optimized, generating recommendations for single change transactions is fast and takes about one second.[2]

[2]All times were measured on a Linux Server using a single Intel Xeon(X5570) processor (2.93GHz) and 8GB RAM

B. Additional Change Properties

There might be cause-effect chains that occur under certain *circumstances* only. To capture cause-effect chains that are bound to certain development activities, we have to combine long-term coupling rules with *change properties* such as size, author, or purpose of a change.

We implemented two such properties: *fixes* and *big changes*. Analyzing commit messages from version archives using a similar approach like Zimmermann et al. [23], we can determine if the applied changes were made to fix a bug *fix transactions*. Transactions changing more than 20% of a file's content are classified as *big changes* with respect to the individual file. Formulas with otherwise low confidence might have high confidence when considering fixing transactions or big transactions only. An example of such *conditional* long-term change coupling rule is given in Section VII.

C. Inner-Transaction Rules

So far, GENEVA extracts long-term coupling rules occurring across multiple change transactions only. Previous work [18], [21] also reported rules that occur within the very same transaction. For comparison purposes, we added an option adding *inner-transaction rules* to adjust formulas support and confidence values.

In this paper, we obtained all results with disabled inner-transaction rules, since we want to highlight the contribution of long-term couplings. In Section VIII-G, however, we will see that inner-transaction rules increase the number of recommendations without sacrificing the precision.

VII. Some Rule Examples

GENEVA uncovers long-term change couplings. Unlike couplings within transactions, long-term change couplings might not be obvious nor directly understandable by looking at the code in one version—simply because they span a longer period of time. To project outsiders, many long-term change rules may come as surprises (which may actually add to their value). Below, we give three basic examples of long-term coupling rules from the *JRuby* project in CTL style.

Long-term couplings in JRuby. The *JRuby* project is based on a large compiler infrastructure defining many compiler interfaces and implementations for these interfaces. One such interface is called `VariableCompiler`. It changed 18 times between 2001 and 2010. Out of these 18 changes, 16 caused a change in `StandardASMCompiler` within 8 days. Each time, both transactions depended indirectly on each other. Considering the complete JRuby project history, the long-term coupling rule has a support value of 16 and a confidence of 0.77.

$$\mathit{VariableCompiler} \Rightarrow \mathsf{EF}\,\mathit{StandardASMCompiler}$$

Although, both files were never committed together, the maintainer of both files is the same. In addition, both classes call each other indirectly: `StandardASMCompiler` uses `BodyCompiler` as interface; it's implementation `BaseBodyCompiler` then references `VariableCompiler`. The coupling although frequent, can only be detected over time—by using an approach like GENEVA.

Test suite changes. Adding functionality to classes often requires new test cases to be added. Such dependencies occur frequently within the same transactions. But there are also cases in which improved or changed test cases unveil problems in classes that might not be directly under test. These cases often span multiple transactions since the newly discovered issues cannot be checked ad hoc:

$$\mathit{MainTestSuite} \Rightarrow \mathsf{EF}\mathit{RubyObject}$$

Nine changes to the `MainTestSuite` made other developers change the class `RubyObject`. The obvious assumption is that the main test suite unveiled new problems in `RubyObject`; the long-term coupling rule connecting both files has a confidence of 0.65.

For JRuby we found 8 rules having a test case as premises with an average confidence of 0.59 and an average support value of 6.25. Note that most of these case could not be detected by compiling the project(s) due to the fact that most test errors are runtime issues.

Fixes vs. changes. Some change couplings occur only under certain circumstances. The following rule has an overall confidence value below 0.5:

$$\mathit{RubyIO} \Rightarrow (\mathsf{EF}\mathit{RubyStructure} \ \wedge \ \mathsf{EF}\,\mathit{Visibility})$$

However, if the changes applied to `RubyIO` are bug fixes, the rule has a confidence of 0.8. In other words, fixes to `RubyIO` imply other changes, while regular changes do not.

For JRuby we found 31 long-term coupling rules that only occur frequently when fixing an artifact. The average confidence of these rules lies at 0.63 with an average support value of 5.4.

All these rules span both space and time, and reveal long-term couplings that GENEVA is the first approach ever to uncover. Deviations from these rules are likely candidates for missing activities—and hence problems.

GENEVA can use additional change properties as coupling conditions to detect change coupling rules that cannot be detected by comparable tools.

VIII. Quantitative Evaluation

After having explored some of the patterns manually, we wanted to know how many such rules exist and how useful these patterns are in practice. The fact that long-term coupling rules are represented in CTL formulas allows *automatic pattern validation* on other change genealogies. Therefore, the accuracy of this validation is our evaluation target: How reliable are these rules and do patterns really occur frequent enough to allow recommendations?

A. Evaluation Subjects

For our quantitative evaluation, we chose four open source projects that had more than two years of project history and were under constant development by more than twenty developers (see Table II). The project histories contain seven to twelve years of active development, more than 1,300 project revisions and more than 1,000 changed files. The number of those files causing long-term couplings varies from project to project. For three out of the four projects GENEVA found over 200 long-term coupling rules with confidence above 0.5 and for two projects nearly 100 long-term couplings with confidence above 0.7.

Table II
EVALUATION SUBJECTS. '#FILES' IS THE NUMBER OF FILES CHANGED WITHIN THE PROJECT HISTORY. THE ROWS '#LTC 0.5' AND '#LTC 0.7' SHOW THE NUMBER OF SOURCE FILES FOR WHICH LONG-TERM COUPLINGS WITH SUPPORT VALUES ≥ 3 AND A CONFIDENCE ≥ 0.5 AND ≥ 0.7 RESPECTIVELY EXIST.

	ArgoUML	Jaxen	JRuby	XStream
history	12 years	9 years	9 years	7 years
transactions	16,481	1,353	11,060	1,683
authors	50	20	66	11
#files	16,658	9,831	15,029	1,188
#ltc 0.5	243	28	232	231
#ltc 0.7	94	10	99	19

B. Exploring Change Genealogy

Table III shows details of the genealogy graphs for the four project histories. The number of vertices equals the number of transactions changing at least one JAVA source file. GENEVA cannot determine dependencies between non JAVA files and thus ignores these files.

For each project, we had to determine an appropriate window size *max_days*. Choosing *max_days* too small will disregard many potential long-term couplings, but choosing it too large will spoil the results by adding a lot of noise. As a first approximation, we therefore used the change genealogy graphs to compute the *median number of days between two dependent transactions* to determine a reasonable value for the window size to be used. Table III shows that the median time gaps between vertices and youngest child (*Median GYC*) vary between four and sixteen days.

C. Predicting Long-Term Couplings

To evaluate whether long-term coupling rules are precise enough to be used for recommendations, we build a long-term coupling pattern prediction model. GENEVA ranks recommendations by their confidence and support values. The top ranked three recommendations are then used as prediction result. The first 10% of each project history define the training period allowing GENEVA to learn common long-term change patterns and rules. For the remaining 90% of transactions, we used GENEVA to predict the top three ranked long-term coupling rules:

Table III
DETAILS OF GENEALOGY GRAPHS FOR THE FOUR OPEN SOURCE PROJECT HISTORIES. OD STANDS FOR OUT DEGREE—THE NUMBER OF EDGES LEAVING A VERTEX. "*Median GYC*"—MEDIAN GAP YOUNGEST CHILD—IS THE NUMBER OF DAYS BETWEEN A VERTEX AND ITS YOUNGEST CHILD. #SUBGRAPHS STATES THE NUMBER OF GENERATED, NON-EMPTY GENEALOGY SUBGRAPHS OBTAINED WHEN USING *Median GYC* AS THE TIME WINDOW SIZE.

	ArgoUML	Jaxen	JRuby	XStream
#Vertices	8,716	1,330	11,055	1,680
#Edges	28,389	3,651	63,613	4,161
Average OD	7.9	6.7	10.1	5.3
Median GYC	16	9	8	4
#subgraphs	1,216	128	2,387	256

1) Let T be the transaction to predict rules for. Further, let CF_T be the set of files changed by T.
2) Remove all files $f \in CF_T$ from CF_T that got deleted by T: $CF'_T = CF_T \setminus \{f \in CF_T | f$ got deleted by $T\}$. Remove all rules that have f as implication premise.
3) Take all CTL rules seen in the past that have file $c \in CF_T$ as implication premise and store as prediction candidates PC_T.
4) Remove all entries from PC_T that have a confidence lower than 0.5 or a support value lower than 2: $P_T = \{pc \in PC_T | \mathit{conf}(pc) \geq 0.5 \ \wedge \ \mathit{support}(pc) \geq 3\}$.
5) Sort P_T by confidence. Rank entities with equal confidence using their support value. Remove all but the top three entities from P_T.
6) Let G_T be the genealogy subgraph for the transaction T using *max_days* (see Table III). Let F_{G_T} be the set of CTL formulas that evaluate to true on G_T.
7) Update support and confidence values of all know formulas and add new rules.

D. Benchmark Model

To illustrate the usefulness of this approach, we compare the prediction measurements with a very basic benchmark model that constantly predicts the top three most changed files, at the prediction point in time. In Table IV, benchmark values are stated behind the GENEVA result in brackets.

E. Precision of recommendations

As performance measurement for our prediction model, we compute its *precision*. A standard metric for the fidelity of classifications, the precision determines the fraction of correctly predicted long-term change coupling rules:

$$\mathit{precision} = \frac{\#\mathit{true\ positives}}{\#\mathit{true\ positives} + \#\mathit{false\ positives}}$$

Table IV shows the prediction results of the described prediction process. The precision of the prediction model lies between 60 and 72 percent—thus, roughly two out of three recommendations correctly predict a future code change that

Table IV
GENEVA PREDICTION RESULTS FOR LONG-TERM COUPLINGS. "AVG RANK OF HIGHEST HIT" DETERMINES THE AVERAGE POSITION THE FIRST VALID RULES CAN BE FOUND AT. ROW 3 SHOWS THE PERCENTAGE OF TRANSACTIONS FOR WHICH GENEVA GAVE AT LEAST ONE RECOMMENDATION. THE LAST ROW SHOWS HOW MANY OF THE GIVEN RECOMMENDATION SETS CONTAINED TRUE LONG-TERM COUPLING RULES ONLY.

	GENEVA				**GENEVA with inner-transaction rules**			
	ArgoUML	Jaxen	JRuby	XStream	ArgoUML	Jaxen	JRuby	XStream
Precision (Benchmark)	0.60 (0.31)	0.70 (0.28)	0.72 (0.58)	0.63 (0.28)	0.66	0.59	0.71	0.67
Avg rank of highest hit (Benchmark)	1.8 (2.0)	1.8 (2.4)	1.9 (2.1)	1.8 (2.1)	2.0	2.0	2.0	2.1
% Transactions recommended	9.2	9.3	32.6	20.7	21.5	17.2	43.8	37.1
% Transactions with only true positives	52.3	68.8	58.0	54.2	48.0	47.8	49.1	43.8

will depend on the current code change within the time frame of *max_days*. This precision is on par with systems like eROSE [18] but clearly outperforms the precision of the benchmark model. Given that the recommendations actually span space *and* time and thus face a far greater challenge, this is a very satisfying result.

Precision is usually accompanied by *recall,* a measure of completeness of our predictions. In our setting, this would mean to evaluate how much of a system's future evolution (as expressed by future long-term couplings) is predictable from its past history. Since the future is determined by so many factors that are completely outside of the domain of our research (and far out of the capabilities of any research), the measure of recall makes little sense in our context.

In our evaluation, two out of three recommendations by GENEVA correctly predict the next activity.

F. Efficiency of recommendations

In GENEVA's recommendations, the average rank position of the highest ranked true positive is between one and two. This means that in case GENEVA gives a recommendation, the first or second recommendation is a hit. Thus, choosing a recommendation list length of more than three would still deliver valid long-term coupling rules at the top of the recommendation list.

GENEVA's recommendations are efficient, placing the correct activity in the top two positions of the ranked list.

G. Adding inner-transaction rules

The right half of Table IV shows prediction results for the enhanced GENEVA tool integrating inner-transaction coupling rules (Section VI-C). With inner-transaction rules, the prediction precision for the projects argoUML and XStream slightly improved, while the precision for Jaxen and JRuby decreased. Overall, though, it seems that the integration of inner-transaction change patterns does not add many coupling rules not known by GENEVA before. The average rank of the highest true recommendation slightly drops but remains stable across all projects. Surprisingly, the number of vertices for which GENEVA gives recommendations increases drastically. Together with a stable to slightly improved precision, we can conclude that adding inner-transaction rules to GENEVA improves the overall results. This also implies that both sets, inner-transactional and long-term couplings, are not subsets of each other. Increasing the number of recommendations without decreasing precision implies that GENEVA added rules that could not be detected within single transactions and vice versa.

Adding inner-transaction rules increases the number of recommendations without sacrificing precision.

H. Threats to Validity

External validity. We only examined the histories of four open-source projects. We cannot claim their development process or project history is representative for other projects. However, we expect projects with a tighter process control to result in more process rules and increased accuracy.

Internal validity. Our approach of modeling dependencies between changes by methods is kept simple on purpose, and is neither necessarily sound nor necessarily complete. As the problem is generally undecidable, a certain amount of imprecision cannot be avoided. Concepts like the coupling between transactions (rather than atomic changes) or the use of time windows may also reduce precision while improving efficiency.

Construct validity. Our evaluation used a standard approach: Learning from the past and checking whether our findings still hold in the future; no manual interpretation was involved that could threaten our findings.

IX. Conclusion and Consequences

In software development, activities are spread across space and time—and yet depend on each other. *Change genealogies* capture these dependencies as long-term couplings between changes and affected artifacts. GENEVA is able to detect such long-term couplings as *temporal rules* that capture key features of the underlying software process. GENEVA predicts code changes that will be applied in future with a precision around 70% (Table IV), which is considerably higher than predicting the most frequently changed files. Being able to predict across space and time shows the potential of change genealogies in predicting software features.

Besides general improvements to performance and scalability, our future work will focus on the following topics:

Dependencies between changes. Currently, we are investigating dependencies between *changes* to allow GENEVA to use a much finer-grained dependency graph to increase prediction precision.

More features. Right now, we express temporal patterns over individual files affected by a change. Rules may also include authors ("Whenever Bob changes something, Alice revises it") or metrics ("If cyclomatic complexity exceeds 0.75, a module will be refactored").

Graph patterns and metrics. Besides temporal rules, we can also search for specific *patterns* in the genealogy graph, such as identifying changes that trigger the most future changes, or the changes with the highest long-term impact on quality or maintainability.

Acknowledgments. Kim Herzig is funded by a Google Research Award "Predicting the Risk of Changes".

References

[1] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip, "Finding failure-inducing changes in java programs using change classification," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14, 2006, pp. 57–68.

[2] J. Wloka, B. Ryder, F. Tip, and X. Ren, "Safe-commit analysis to facilitate team software development," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 507–517.

[3] I. I. Brudaru and A. Zeller, "What is the long-term impact of changes?" in *RSSE '08: Proceedings of the 2008 international workshop on Recommendation Systems for Software Engineering*, 2008, pp. 30–32.

[4] 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, no. 10, pp. 1394–1408, 2009.

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

[6] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference*, 2003, pp. 128–137.

[7] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 308–318.

[8] O. Alam, B. Adams, and A. E. Hassan, "Measuring the progress of projects using the time dependence of code changes," in *ICSM*, 2009, pp. 329–338.

[9] K. S. Herzig, "Capturing the long-term impact of changes," in *ICSE '10 Ph.D. Symposium: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 393–396.

[10] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 215–249, 1998.

[11] V. Curcin, M. M. Ghanem, and Y. Guo, "Analysing scientific workflows with Computational Tree Logic," *Cluster Computing*, vol. 12, no. 4, 2009.

[12] L. Wen, J. Wang, W. M. Aalst, B. Huang, and J. Sun, "A novel approach for process mining based on event types," *J. Intell. Inf. Syst.*, vol. 32, no. 2, pp. 163–190, 2009.

[13] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 345–354.

[14] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 295–306.

[15] E. P. F. Chan, "Temporal-logic queries," *Computer Aided Verification*, vol. 12, no. 4, pp. 450–463, 2000.

[16] A. Gurfinkel, M. Chechik, and B. Devereux, "Temporal logic query checking: A tool for model exploration," *IEEE Transactions on Software Engineering*, vol. 29, pp. 898–914, 2003.

[17] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *ICSM '98: Proceedings of the International Conference on Software Maintenance*, 1998, pp. 190–198.

[18] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 563–572.

[19] A. T. T. Ying, "Predicting source code changes by mining revision history," 2003.

[20] B. Fluri, H. C. Gall, and M. Pinzger, "Fine-grained analysis of change couplings," in *SCAM '05: Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, 2005, pp. 66–74.

[21] 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 18th International Conference on Software Maintenance (ICSM)*, ser. ICSM 2010, 2010.

[22] B. Dagenais and L. Hendren, "Enabling static analysis for partial Java programs," in *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2008, pp. 313–328.

[23] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '07, 2007.