Journal of Software Engineering and Applications, 2013, 6, 157-167
http://dx.doi.org/10.4236/jsea.2013.64020 Published Online April 2013 (http://www.scirp.org/journal/jsea)

IMPEX: An Approach to Analyze Source Code Changes on Software Run Behavior

David Nemer

School of Informatics and Computing, Indiana University, Bloomington, USA.
Email: dnemer@indiana.edu

Received February 28th, 2013; revised March 31st, 2013; accepted April 8th, 2013

ABSTRACT

The development of software nowadays is getting more complex due to the need to use software programs to accomplish more elaborated tasks. Developers may have a hard time knowing what could happen to the software when making changes. To support the developer in reducing the uncertainty of the impact on the software run behavior due to changes in the source code, this paper presents a tool called IMPEX which analyzes the differences in the source code and differences on the run behavior of two subsequent software versions, in the entire repository, demonstrating to the developer the impact that a change in the source code has had on the software run, over the whole software history. This impact helps the developers in knowing what is affected during execution due to their changes in the source code. This study verifies that the software runs that are most impacted by a given change in the source code, have higher chances in being impacted in the future whenever this part of the code is changed again. The approach taken in this paper was able to precisely predict what would be impacted on the software execution when a change in the source code was made in 70% of the cases.

Keywords: Change Impact; Change Prediction; Quality Assurance; Software Development

1. Introduction

As the use of computer software is demanded for realizing more and more complex tasks, the development of these software are also getting more complex each and every day, making the life of the developer a bit harder. This paper attempts to help the developers by proposing an approach to support them in knowing, before making any changes in the software, what can be impacted when something is modified in that particular part of the code.

Imagine a new developer at a software development project, she needs to get used to the new environment and understand the project's code. Resources at hand, such as the documentation, can be a good help but it may be out dated. A personal help, e.g. project manager, could also aid her, but the person won't always be with her. Since she needs to go ahead with the project development, she might not be aware of how dangerous it could be to change a certain line of code. In other words, she may not know how risky her change might be regarding to the rest of the program.

Not only to new developers, but experienced developers in a project must also deal with this sort of situation, since it is very unlikely for them to know the whole source code.

To support the developer in reducing the uncertainty of the effects (or impact) of the changes, this paper attempts to predict what parts of the software system is impacted during execution, resulting from changes in the source code. This "impacted" means, making a change in some part of the code, may also affect other parts while execution. This knowledge is important for software maintenance as the analysis, which can be applied either before or after changes are made to the software system. It can also provide a valuable guide to the software engineers. Applying our approach before a change is made, allows a software engineer to determine what components may be affected by the change and gauge the cost of the change.

These issues presented above basically drive this paper and shape the approach that is detailed in the following.

2. Background

The main goal of this paper is to measure the impact that changes in the source code have on the software during execution and aggregate this information, so predictions can be made on what will be impacted when a certain change is made. For such, the proposed tool, IMPEX, observes what was changed, between two revisions of the software, in the source code, and also see the difference that these changes caused during run-time. To get the differences in the source code, as well as in the execution, IMPEX relies on two third party tools. Javalanche [1] was used in order to aid IMPEX in getting the run behavior and comparing two different executions of the software. When dealing with comparing two versions of the source code, IMPEX counted with API-level Code Matching. In the following, this section presents and details the tools API-level Code Matching and Javalanche, later, the sections Approach and Evaluation detail how these tools fit with IMPEX.

2.1. Javalanche

Javalanche is a framework originally developed to support mutation testing of Java programs [1,2]. Mutation testing measures the adequacy of a test suite by seeding artificial defects (mutations) into a program. Javalanche was primly used to support checking the invariant violations to detect mutations [1]. It also assesses the impact on invariants and impact on code coverage [2].

In order to analyze two different version of a software and identify the difference of execution behavior between them, IMPEX uses Javalanche. Originally, Javalanche produces mutated versions of a software by inserting mutations in these versions, and then compare their run behavior with the original version to assess the impact of these mutations. Since mutations are considered to be changes in the software, IMPEX counts with the support of Javalanche to identify differences in the run behavior of two software revisions which were not changed by adding mutations but by actual development changes. This functionality of Javalanche had to be modified and adapted in order to assess the impact of real development changes in the code. The differences of mutation changes and development changes are later explained in this section.

2.1.1. Javalanche and the Impact of Equivalent Mutants

Javalanche creates mutated versions of a software by adding mutation in the code, and later checks whether the test suite is capable to detect theses mutations in the code of the mutated versions. The idea is to have the test suite detecting all the mutations, but if a mutation is not detected by a test suite, this usually means the test suite is not adequate [1]. With the capability of a test suite finding mutants, test managers can improve their test suites such that they detect these mutants. In order to produce mutations, Javalanche uses a small set of operators, it relies on replacing numerical constant, negating jump conditions, replacing arithmetic operator and omit method calls. The **Table 1**, which was extracted from David Schuler *et al.* [1], represents the mutations just mentioned.

An example of a mutated code is given in **Figure 1**. It might be the case that even though a mutant is seeded, the program's semantics remains unchanged, hence, it can't be noticed by any test. These sorts of mutants are called equivalent mutants, and they need to be removed manually, which is a tedious task, and they have less impact on execution. Grün *et al.* [2] propose an approach based on their experiments, focusing mainly on mutations that alter the dynamic control flow, they tend to be less likely to be equivalent. The higher is the impact, the lower is the chance of equivalence.

The impact of a mutation can be assessed by checking the program state at the end of a computation, as tests do. However, it can also be assessed while the computation is not complete. In particular, changes in program behavior between the mutant and the original version can be measured. One aspect that is particularly easy to measure is control flow [1]: If a mutation alters the control flow of the execution, different statements would be executed in a different order—an impact that is easy to detect using standard coverage measurement techniques.

Grün *et al.* [2] tested the relationship between coverage and non-equivalence by developing a program that computed the code coverage of a program, and integrate it into the Javalanche framework.

Table 1. Javalanche mutation operators.

Replace a numerical constant	Replace a numerical constant X by X + 1, X − 1, or 0.
Negate jump condition	Replace a conditional jump by its counterpart. This is equivalent to negating a conditional statement in the source code.
Replace arithmetic operator	Replace an arithmetic operator by another arithmetic operator, e.g. + by −.
Omit method calls	If the method has a return value, a default value is used instead, e.g. X = Math. random () is replaced by x = 0.0.

```
// Original code:
if (x == 1)
{
   x++
};

// Mutated code:
if (x != 1)  // != is the mutation
{
   x++
};
```

Figure 1. An example of a mutation.

To measure the impact of a mutation, each program's mutation and every test case is computed, as well as, the statement coverage (number of times a statement is executed) of the original program. The coverage of the original execution is compared with the coverage of the mutated execution which results to a coverage difference. As an impact measure, Grün *et al.* chose the number of classes that have different code coverage. This measure is motivated by the hypothesis that a mutation that has non-local impact on the coverage is more likely to change the observable behavior of the program. Furthermore, it is assumed that mutations that are undetected despite having impact across several classes, to be particularly valuable for improving the test suite. As the test suite indicates inadequate testing of multiple classes at once, it also shows that the higher the impact of a mutation, higher the chance of it being non-equivalent.

2.1.2. Javalanche and Impact of Changes

Even though Javalanche originally evaluates the impact of mutations, it also can evaluate the impact of code changes. This paper was successful in adapting Javalanche to produce and compare the run traces of two versions of a source code. In section Approach, it is explained in more details how the adaptation of Javalanche was done and how it supported IMPEX in order to fulfill this paper's goal.

Every mutation can be considered a change in the code, but not every change in the code is a mutation [1]. Javalanche uses a small set of operators to produce mutations, it relies on replacing numerical constant, negating jump conditions, replacing arithmetic operator and omit method calls, as represented in **Table 1**.

As noticed, Javalanche, simply replaces a code with a mutation. It doesn't add new lines in the code, it doesn't implement new functionalities for example, these changes are actual code changes, and heavily exceed the modifications proposed by Schuler *et al.* [1]. Even though development changes differ from mutations, Javalanche is still able to compute the differences between two version's runs, in other words, it is able to tell us what part of the code had a different behavior compared to the previous execution. When adding a mutation in the software, Javalanche keeps track of this mutation and it is able to tell where the mutation was placed, when it comes to actual development changes, Javalanche is not aware of the changes made because it was not done by it, therefore, the differences in the source code of the two versions are needed, to see what was changed there, so they can be assumed to be the cause of the different behavior.

2.2. API-Level Code Matching

To get actual changes made in the source code, two versions of the source code have to be compared so differences between them can be identified. For this task, IMPEX is supported by the API-level Code Matching, which is a tool developed by Kim *et al.* [3]. The changes identified by API-level Code Matching will be later related to the changes obtained by Javalanche during execution of the software. With that, we can identify what was changed in the source code and what was changed during execution.

The API-level Code Matching maps code elements in one version of a program to the corresponding code elements in another version, it outstands itself among the others similar tools, because it overcomes two main limitations: first, existing tools cannot easily disambiguate among many potential matches or refactoring candidates, and second, it is hard to use the results of these tools for software engineering tasks due to an unstructured representation of the results [3]. The API-level Code Matching represents structural changes as a set of high level change rules, automatically infers likely change rules and determines method level matches based on these rules.

Matching code elements between two versions of a software is the underlying basis for various software engineering tools, such as IMPEX. For instance, version merging tools identify possible conflicts among parallel updates by analyzing matched code elements, regression testing tools prioritize or select test cases that need to be rerun by analyzing matched code elements, and tools which compare source code of different versions of a program, use code matching to deal only with actual changes in the code, not identifying refactoring and restructuring as changes that affects the logic of the program, this is the example which IMPEX fits.

The API-level Code Matching approach automatically infers likely changes at or above the level of method headers, and uses this information to determine method level matches, also it represents the inferred changes concisely as first-order relational logic rules, each of which combines a set of similar low-level transformations and describes exceptions that capture anomalies to a general change pattern [3].

The API-level Code Matching tool showed to be an efficient way to aid our approach in optimizing the process of comparing the source code of two versions of a program. It is quite simple to get differences between two codes, but it is important to take into account refactoring and restructuring, which are changes, but they are not changes that affect the logic and the behavior of the software. This is what the tool presented by Kim *et al.* works at, matching code so these changes are not taken for granted.

3. Approach and the Tool IMPEX

The main goal of this study is to measure impact of code changes on the executional behavior of the program.

Once being able to measure this runtime impact, IMPEX checks whether source code changes having large runtime impact are more risky to be impacted or not in the future. To accomplish the first step of impact measuring, the changes done in the source code are mapped to what was affected during execution. These data are identified by comparing two subsequent revisions of a software, getting the differences in the source code between these two revision, as well as what was changed during execution. In order to see what each change in the source code really impacts, IMPEX goes over the whole repository and map each of them to what was affected during execution. IMPEX consists in five main steps, and they are all automated. IMPEX is illustrated in **Figure 2** and the steps are: 1) obtaining the revisions (checking out); 2) comparing the source code between the two revisions; 3) building the revisions; 4) comparing the run traces between these two revisions and 5) mapping the source code differences with the run traces differences.

The first step of IMPEX is to check out two consecutive revisions, for example $revision_N$ and $revision_{N+1}$. In the second step and fourth step, the tool compares both revision's source code and execution traces, and map their differences. The comparison process is done with the support of the external tools presented before:

- API-level Code Matching, that handles differences in the source code.
- Javalanche, which is in charge of producing the run traces and the differences between them.

API-level Code Matching directly compares the versions of the source code which are checked out from the repository, and, since Javalanche works on Java byte code, step three is needed, which is having the revisions compiled (built). In order to have an automated compiling process, IMPEX currently supports Ant and/or Maven, which are automated building tools.

The fifth step consists in mapping changes done in the source code to what was affected during execution. Each change in the source code is mapped to the change of behavior on the run, so it can be assumed that the changes made in the source code were responsible by impacting the different behavior. For example, when comparing the source code of $revision_N$ and $revision_{N+1}$ IMPEX would get differences in the source code of Class1 and Class2, while Class3 and Class4 would be the classes that had a different behavior during execution. It is presumed that the changes in the source code of Class1 and Class2 caused a different run behavior in Class3 and Class4. In other words, the changes in Class1 and Class2 impacted the run of Class3 and Class4.

All of the five steps are done automatically in the whole repository. The outcome of IMPEX is a matrix which will help the developer see the impact that the source code changes have on the execution, thus showing to the developer which parts of the source code can impact during execution. This matrix produced is defined on this paper as the prediction model. IMPEX produces the prediction model in three levels: method, class and package level. In **Table 2** there is an example of the prediction model produced:

Table 2 shows an example of the prediction model in the class level. It can be explained as following, in the whole repository, the source code in Class1 was changed 10 times, there were 5 times (five differences of revisions) that the source code in Class1 was changed and also the run traces in Class4 had a difference. The prediction model produced is explained in more details in Section Data Gathering. As explained before, every step is automated, but the user still needs to manually specify details such as:

- The repository's address.
- Type of repository (http, file or svn).
- Whether the version checked out is built by Ant or Maven, and indicates the location (path) of the build file, which is build.xml for Ant, pom.xml for Maven2 and project.xml for Maven1.
- Javalanche attributes, such as class path and test suite location.
- API-level Code Matching attributes, such as location of the source code.

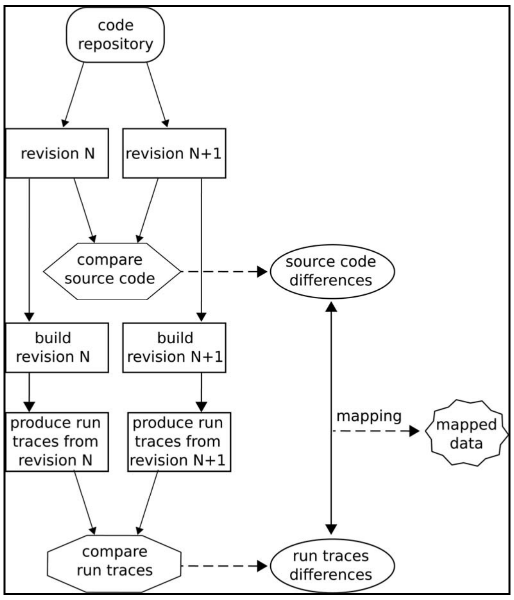

Figure 2. Diagram of the steps of IMPEX.

Table 2. Example of prediction model produced by IMPEX.

Changes on source code detected	Changes on run traces detected		Number total changes on source code
	Class3	Class4	
Class1	8	5	10
Class2	6	9	11

Since Javalanche is also automated and plays a key role in IMPEX, the tool ends up depending a lot on the consistency of the revisions obtained, in other words, the revisions needs to be buildable, also, Javalanche produces the difference of run traces based on the execution of the test cases of each revision, therefore, the test cases also need to be executed flawlessly. Sometimes errors during the building and testing processes, due to internal revision issues, can happen, thus precluding the production of run traces. If this is the case, IMPEX simply skips problematic revisions to the next computable (buildable and testable) revisions. Another main issue that IMPEX deals is having two consecutive revisions with different set of test cases. It has to map differences in the run traces caused by changes in the source code, and not changes caused due to modification in the test cases, therefore, IMPEX limits Javalanche to compare the run traces produced by the same test cases contained in both revisions. What IMPEX does is to check which test case was modified, added or deleted, between two revisions, and then, delete the traces produced by them, thus comparing the traces produced by unmodified test cases.

Another issue IMPEX deals with is nondeterministic behavior. When producing traces from execution, some software revisions might have nondeterministic behavior, which leads to producing different run traces in different executions. To overcome this issue, IMPEX executes several runs of a single revision of the software, and compares the traces produced by every execution. The traces produced by Javalanche are on the method level. If there are different methods between two set of traces from the same revision, IMPEX simply excludes the traces of such methods, since they had a different behavior due to nondeterministic behavior, and not due to actual changes in the source code.

The pseudo code of IMPEX illustrates in a programmatically way the steps mentioned above, it is in the following **Figure 3**.

3.1. Adapting Javalanche

Javalanche is the responsible for producing the run traces from the built revisions. Originally, Javalanche produces mutations of a revision, and calculates the impact of the mutations on the code coverage by comparing an unmutated run of the test suite with the mutated run of the test suite. Javalanche had to be adapted in order to compare the run traces of two actual revisions, instead of the traces of mutations. When comparing traces of a mutated version and the original version of the software, Javalanche identifies where the run traces from both versions are, but when comparing two revisions of the same software IMPEX passes to Javalanche where the traces are stored.

To manage the nondeterministic behavior of the same revision, Javalanche was adapted in a way that it produces traces of different executions of such revision. Instead of having it comparing the runs of traces from two different revisions, IMEPX indicates to Javalanche where the two set of traces of the same revision are stored, and have Javalanche comparing them, as if it was comparing two different revisions. IMPEX does that to every set of execution traces, if the Javalanche's comparison is not able to find any method that behaved differently, that means the execution of the revision doesn't show nondeterministic behavior, otherwise, if Javalanche finds methods in the comparison, they are considered to be nondeterministic, therefore, IMPEX excludes them. These methods had a difference due to nondeterministic behavior and not because of actual changes in the source code.

When adding a mutation in the software, Javalanche keeps track of this mutation and it is able to tell where the mutation was placed. When it comes to actual development changes, Javalanche is not aware of the changes made because it was not done by it, therefore, IMPEX computes the differences in the source code of the two versions, to see what was changed there, so it can be assumed that these changes caused the different behavior. To support IMPEX in finding the changes in the source code, it counts on the API-level Code Matching tool [3] explained in the following.

```
begin
   for (All Revisions in the Repository) do
      revision_pre = check out revision N;
      revision_post = check out revision N+1;
      source_differences = Api-level Code Matching(revision_pre, revision_post);
      if (source_differences <> empty) then
         revision_pre_built = build(revision_pre);
         revision_post_built = build(revision_post);
         run_traces_differences = Javalanche(revision_pre_built, revision_post_built);
         if (run_traces_differences <> empty) then
            matrix_with_mappings.map(source_differences, run_traces_differences);
         end if
      end if
   end for
   return matrix_with_mappings
end
```

Figure 3. Our tool's algorithm pseudo code.

3.2. Adapting API-Level Code Matching

The API-level Code Matching is the tool that helps this approach in getting the differences between two source codes. API-level Code Matching doesn't directly compare two source code in order to get the differences between them. What it does, is matching code elements or identifying structural changes between two versions of a program.

The API-level Code Matching provides an automated way to parse and break the code into several small parts, allowing IMPEX to check if there are any code differences between these small parts from the two revisions. The functionality used from the API-level to support IMPEX, parses the entire code into a list of Method objects.

The Method objects have all the details of the methods. It has an attribute called body which stores the content of the method, and it disregards any unnecessary text such as comments and empty lines, so what is in the body attribute is pure code. Having the code broke down in Method objects helps IMPEX to deal with code refactoring issue, because no matter where the method is placed, it will always be compared (if it exists in both revisions) with the matching other. The API-level Code Matching doesn't use only the signature of a method to identify the other match, it is able to represent structural changes as a set of high-level change rules, inferring likely change rules to determine the method matches based on the rules [3]. So if just a method name is changed, the tool is still able to find a match with a different name. Also, these rules are applied in the body attribute of the matched methods, to identify structural changes, which are not actual code changes.

The API-level Code Matching doesn't take into account what is outside the methods, such as static variables. So IMPEX wraps what was left outside the methods, and add into the list of Method objects as a special type of method, internally identified as such, so when the comparison takes place, IMPEX is able to identify it as a Method object containing class body parts.

3.3. Data Gathering

The final step of IMPEX is to map the source differences with the run differences. After mapping all revisions in the repository, it produces several matrixes, which is defined as the prediction models. These prediction models are aimed to predict what will be impacted in the future, based on the impact that changes in source code had on execution over the whole repository history. Since Javalanche and API-level Code Matching present their results on the method level, IMPEX is able to use their results to gather information on package and class levels as well, but not deeper in the code. This paper determined a set of metrics that defines such impact. The metrics chosen are:

- Number of times a method has a different execution behavior caused by a change in the source code of a method.
- Number of times a class has a different execution behavior caused by a change in the source code of a class.
- Number of times a package has a different execution behavior caused by a change in the source code of a package.

Impact is defined as such, let the repository to consist of revisions $R_1, \cdots, R_k$. For revisions R_N and R_{N+1}, let $c_1, \cdots, c_n$ be parts of the software which differ in R_{N+1} from R_N in the source code. Further, let $c'_1, \cdots, c'_m$ be all parts of the software which is executed in R_{N+1} and R_N. Then for revisions R_N and R_{N+1}, we define the function $f_{N,N+1} : R_N \to \{0,1\}^m$ which takes $c_i \mapsto (x_{i1}, \cdots, x_{im})$ such that $x_{ij} = 0$ if c'_j doesn't differ in R_{N+1} from R_N; and $x_{ij} = 1$ if c'_j differs from R_N, for $i = 1, 2, \cdots, n$ and $j = 1, 2, \cdots, m$. Then the prediction model can be defined as having the sum in each row: $\sum_{N=1}^{k-1} f_{N,N+1}(c_i)$ for each $i = 1, 2, \cdots, n$. An example of prediction model (on class level) is showed in **Table 2**.

The prediction models define the metrics to measure the impact that changes in the source code have on the software runs. Also, they are the basis to predict future impacts based on changes in the source code, this is further explained in section 0.4. For the evaluation and experiments conducted, the package and class levels were considered. The method level was disregarded because, during the experiments, while mapping the entire repository, the prediction model on the method level didn't show enough impacts to compute any metrics or make any predictions. The method level is too fine grained, for both projects, JodaTime [4] and Jaxen [5], the methods in the source code could only be mapped, at most, twice to a method in the run. A real example from Jaxen is shown below in **Table 3**.

4. Evaluation

In order to predict what will be impacted during execution when a change in the source code is done, IMPEX produces a matrix, called the prediction model, with the impacts, which are numbers of times a change in the source code impacted a given part of the run. This impact indicates that, the higher is the number of times a source code change impacted some part of the run, the greater is the probability of this change impacting the same part of the run again in the future. The idea is to provide a model to the developer so she can see what a certain change will impact, and based on the impact numbers, she can see what parts have a higher probability to be impacted, so she knows beforehand what her changes can affect and where she should take actions.

To validate such idea, the prediction model is checked whether it is able to predict what will be affected based on the impact numbers, in other words, if the impacted parts of a run, are actually the parts that have a higher probability to be impacted. Random code modifications

Table 3. Part of Jaxen's prediction model on method level.

Source code	Changes on run traces detected		Number total changes on source code
	registerFunction	getInstance	
registerFunction	2	0	2
getInstance	0	1	1

registerFunction belongs to org. jaxen. Simple Function Context and getInstance belongs to org. jaxen. XPath Function Context.

can't be done to produce different versions to be tested on the prediction model, because they won't represent real changes that the development team of the project is used to do. Therefore, the following scheme is proposed in order to evaluate the predictions: IMPEX is used to produce the prediction model based on two thirds of the revisions in the repository, and the last one third of the repository is used as "future" revisions. Then, the prediction model can be tested whether it is able to predict what will be impacted in these "future" revisions, based on the changes done in the source code. **Figure 4** illustrates the evaluation scheme.

Javalanche and API-level Code Matching present their results on the method level, but IMPEX is able to use their results to gather information also on package and class levels as well. So the prediction model is produced to make predictions in three levels: method, class and package. In the JodaTime [4] and Jaxen [5] projects, the methods in the source code were impacting, at most, twice a method in the run over the whole repository history, and they were still very seldom. Therefore in this evaluation, the method level is not taken into account since it is too fine grained and doesn't provide enough impacts to make any predictions.

It could be the case that code differences don't impact the parts of the run which had a higher probability, but this papers still sees how far off is the actual comparison to the prediction model, and still makes a prediction that is not entirely precise, but would still give the developer an idea of what could possibly be impacted. The impact probability is defined as such:

For a given revision *R*, to calculate the impact probabilities, let $A_1,\cdots,A_n$ be parts of the software that had changes in the source code. Further, let $B_1,\cdots,B_m$ be parts of the software that had changes during runtime. Then for each A_i, we have a function $f_i : R \to \left(\mathbb{N}\cup\{0\}\right)^m$ where $\mathbb{N}$ represents the natural numbers, taking $A_i \mapsto \left(x_{i1},\cdots,x_{im}\right)$ such that x_{ij} represents the number of times A_i impacted B_j, for $i=1,2,\cdots,n$ and $j=1,2,\cdots,m$.

So for a given A_i and B_j, the probability that A_i impacts B_j is $\frac{x_{ij}}{\tau_i}$ where τ_i is the total amount of times

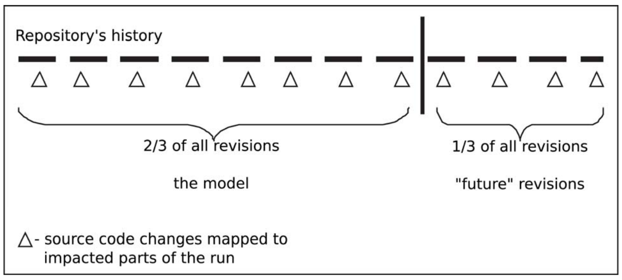

Figure 4. Evaluation scheme.

the source code of A_i was changed over the entire repository history.

The parts with changes in the source code, ideally, should impact the parts of the software execution which have a higher probability. If this always happens, it can be assured that the prediction model produced from the two thirds of the repository was able to predict precisely, every impacted part of the software run from the last one third of the repository. But this is not how it always goes, there may be cases that parts with changes in the source code impact parts of runs that have a low probability in the prediction, thus the need to check how well the prediction model is able to predict these impacted parts.

Each comparison of two consecutive revisions returns, if there is a difference, a set of changes in the source code and a set of differences in run behavior. The prediction is defined as such: $\varpi\left(\vartheta(c)\to\vartheta(c')\right)$, where $\vartheta(c)$ is a set of source code changes and $\vartheta(c')$ is a set of differences in the execution behavior.

We need to verify if the differences in execution behavior in $\vartheta(c')$ impacted by $\vartheta(c)$ are the ones that the prediction model predicted $\vartheta(c)$ to impact, if not, to check how far off the prediction from the prediction model is. In the prediction model, every change in the source code is mapped to a difference of execution behavior with an impact value: $c_i \mapsto \left(x_{i1},\cdots,x_{im}\right)$ such that x_{ij} is the impact value of c_i on $c'_1,\cdots,c'_m$. To compare the prediction with the prediction model, the impact values of c_i on $\vartheta(c')$ are added and subtracted from the sum of the impact values which have the highest probability to be impacted. The comparison can be defined as such, for each c_i in $\vartheta(c)$,

$\sum_{k=1}^{z}\left(X_{ik}\right)-\sum_{j\in\Im}\left(x_{ij}\right)=\Im\varpi$ such that *z* is the number of differences of run behavior in $\vartheta(c')$, *X* is the impact value of the difference in the software behavior which has the highest probability to be impacted by c_i, $\Im$ is the set of indices of $c'_i \in \vartheta(c')$, and *IP* is the result of this subtraction, which will be defined as impact punishment. The smaller is *IP*, closer is the prediction to be according to the prediction model. When *IP* is equal to 0, which is the smallest number it can be, it means that the prediction model was 100% precise in predicting what would be impacted in the prediction, in other words, the source code changes c_i impacted the $\vartheta(c')$, which are the set of differences in execution behavior with the highest probability to be impacted.

IP are not always equal to 0, in other words, the prediction is not always perfect. There may be times that these differences won't be 0, therefore this papers experiments this evaluation to check if the differences are punished way too far from 0, which means that the prediction model cannot be validated, thus predicting anything, or if the differences of impacts are really close to 0, which can be considered as a good prediction model.

5. Experiments and Results

In this section we present the results obtained by applying our approach in two software projects JodaTime [4] and Jaxen [5].

5.1. JodaTime

JodaTime is an open source project, which provides a quality replacement for the Java date and time classes. The design allows for multiple calendar systems, while still providing a simple API. The "default" calendar is the ISO8601 standard which is used by XML. The Gregorian, Julian, Buddhist, Coptic, Ethiopic and Islamic systems are also included, and we welcome further additions. Supporting classes include time zone, duration, format and parsing.

When applied approach presented in this paper to JodaTime, the project had 1.480 revisions in its repository, which the first 320 revisions were not supported by an automated building tool. Therefore, the prediction model (first two thirds of the repository) was based on the data obtained from revision 321 to revision 1100, and the "future" revisions (last one third of the repository) that were used to validate the prediction model were the revisions from 1101 to 1480.

Evaluating the prediction model against the "future" revisions obtained, on the package level, 72.41% of the predictions without any impact punishment, in other words, our approach perfectly predicted what would be impacted in 72.41% of the predictions. Going up to the predictions that had up to 10 impact punishment, which can still be considered a good prediction; our approach predicted 80.17% of the predictions. The results on package level are detailed in **Table 4** and **Figure 5**.

On the class level, our approach was able to predict 25.84% of the prediction without any impact punishment. The predictions that had up to 10 impact punishment, our approach was capable of predicting 56.18% of the predictions, since the class level is a more fine grained level than the package, going up to 20 impact punishment still would give us a good prediction, and our approach was able to predict 79.78%. The results on class level are detailed in **Table 5** and **Figure 6**.

Table 4. Results from JodaTime on package level.

Impact Punishment	Predictions
0	84
1 - 10	9
11 - 20	11
21 - 40	8
41 - 100	3
>101	1

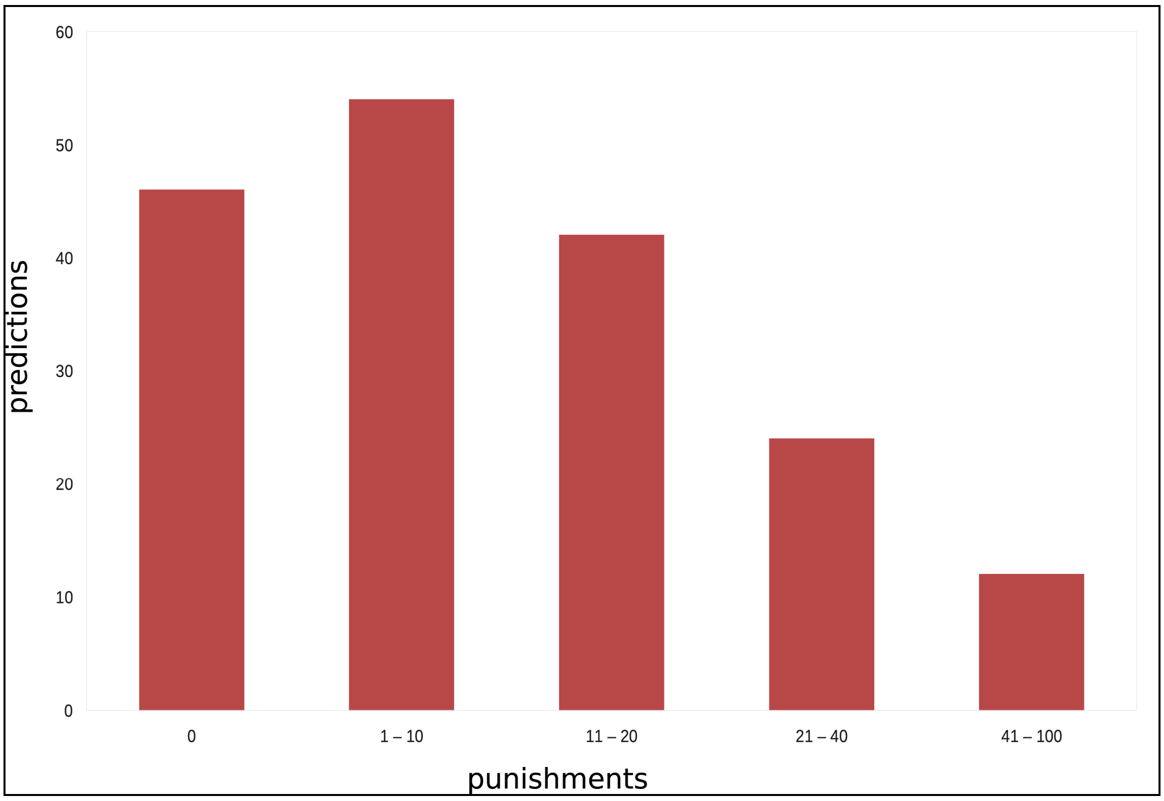

Figure 5. Results from JodaTime on package level.

It is interesting to mention that the predictions, on the package level, which had 0 impact punishment, had its packages from the source code impacting an average of 2.27 packages during execution, and on class level, it had classes from the source code impacting an average of 6.93 classes during execution. This shows that the package, and class, changes didn't only impact itself during run time, but also other packages, and classes, and our approach was still able to predict the impact of them in other packages, and classes. As the total average, on package level, 2.50 packages were impacted, while on class level, 14.01 classes were impacted during execution. The larger is the average of classes and packages impacted, the higher is the probability to get impact differences (impact punishment), this is why going up to 20 impact punishment on class level can't be considered a bad prediction.

Since the results on JodaTime are a bit spread out, we grouped the number of predictions based on the most significant Impact Punishments. In **Figures 5** and **6**, we have on the x-axis the Impact Punishments grouped, and on the y-axis the amount of predictions that had that certain impact punishment.

Table 5. Results from JodaTime on class level.

Impact Punishment	Predictions
0	46
1 - 10	54
11 - 20	42
21 - 40	24
41 - 100	12

5.2. Jaxen

Jaxen is an open source XPath library written in Java. It is adaptable to many different object models, including DOM, XOM, dom4j, and JDOM. Is it also possible to write adapters that treat non-XML trees such as compiled Java byte code or Java beans as XML, thus enabling you to query these trees with XPath too.

When the experiments took place, Jaxen had 1.350 revisions in its repository, which the first 380 revisions were not supported by an automated building tool. Therefore, our model (first two thirds of the repository) was based on the data obtained from revision 381 to revision 1027, and the "future" revisions (last one third of the repository) that were used to validate the prediction model were the revisions from 1028 to 1350.

Evaluating Jaxen, we got good results, comparing the prediction model against the "future" revisions we obtained, on the package level, 83.70% of the predictions without any impact punishment, in other words, our approach perfectly predicted what would be impacted in 83.71% of the predictions. Going up to the predictions

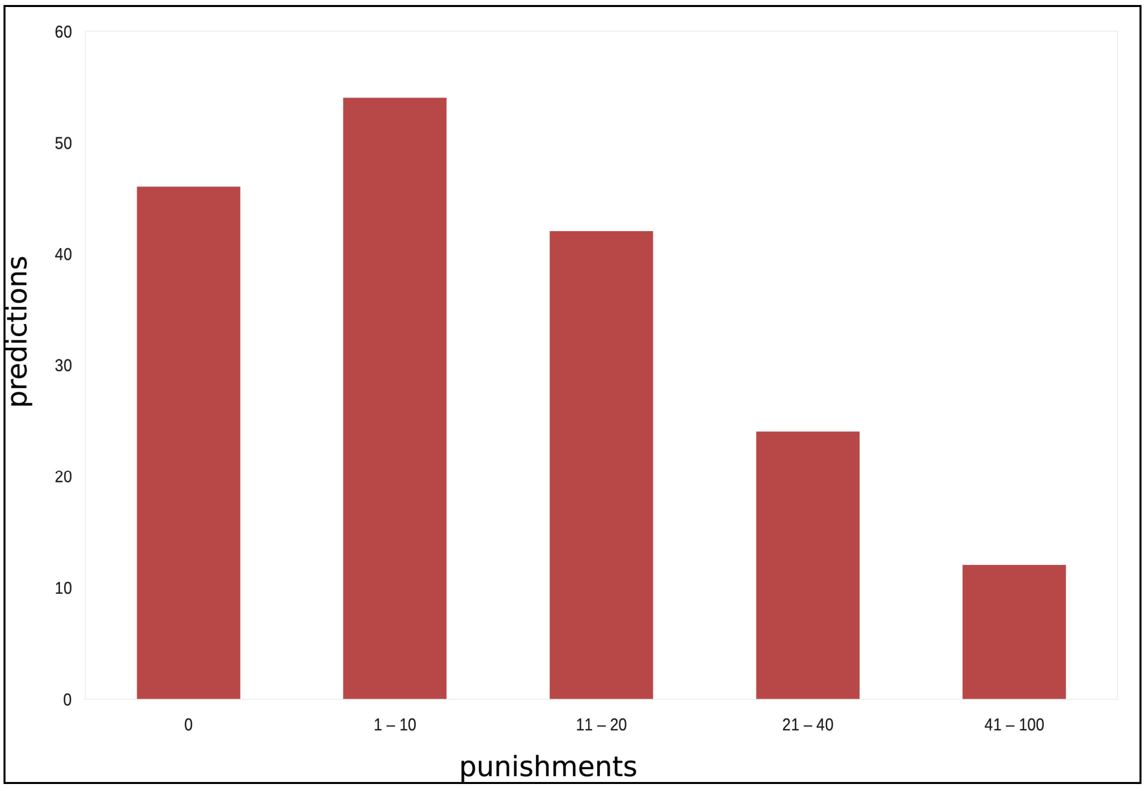

Figure 6. Results from JodaTime on class level.

that had up to 10 impact punishment, which can still be considered a pretty good prediction; our approach predicted 100% of the predictions. The results on package level are detailed in **Table 6** and **Figure 7**.

On the class level, the results still look good as well; our approach was able to predict 77.71% of the prediction without any impact punishment. The predictions that had up to 5 impact punishment, our approach was capable of predicting 98.73% of the predictions, and going up to 15 impact punishment give us a prediction of 100%. The results on class level are detailed in **Table 7** and **Figure 8**.

Table 6. Results from Jaxen on package level.

Impact Punishment	Predictions
0	77
4	7
6	4
8	2
10	2

Table 7. Results from Jaxen on class level.

Impact Punishment	Predictions
0	122
1	14
2	8
3	3
4	3
5	5
15	2

Differently than JodaTime, the predictions with Jaxen, on the package level, which had 0 impact punishment, had its packages from the source code impacting an average of 1.04 packages during execution, these results shows that the changes made in the source code of a package, practically only impacted itself during the execution. On class level, it had classes from the source code impacting an average of 3.02 classes during execution. On the class level, we can't deduce the same thing as on the package level, since classes also impacted other classes and not only themselves. As the total average, on package level, 1.53 packages were impacted, while on class level, 3.38 classes were impacted during execution.

6. Conclusions

This paper had as its main motivation to help the developer in knowing beforehand, what would be impacted if she changed something on the code, as well as inform her how sure that impact could take place. To do so, we developed a tool, IMPEX, and proposed an approach that learns from the history of a software repository and predicts what is most likely to be impacted in the future. Not only the prediction, but our approach proposed a set of metrics that would allow us to identify the probability of that impact to happen.

Our empirical results show that our approach is able to make predictions, even if when it is not 100% precise, it still can make predictions with remarks. Conducting the experiments, we were also able to have an idea about the architecture of the software projects: Jaxen results showed

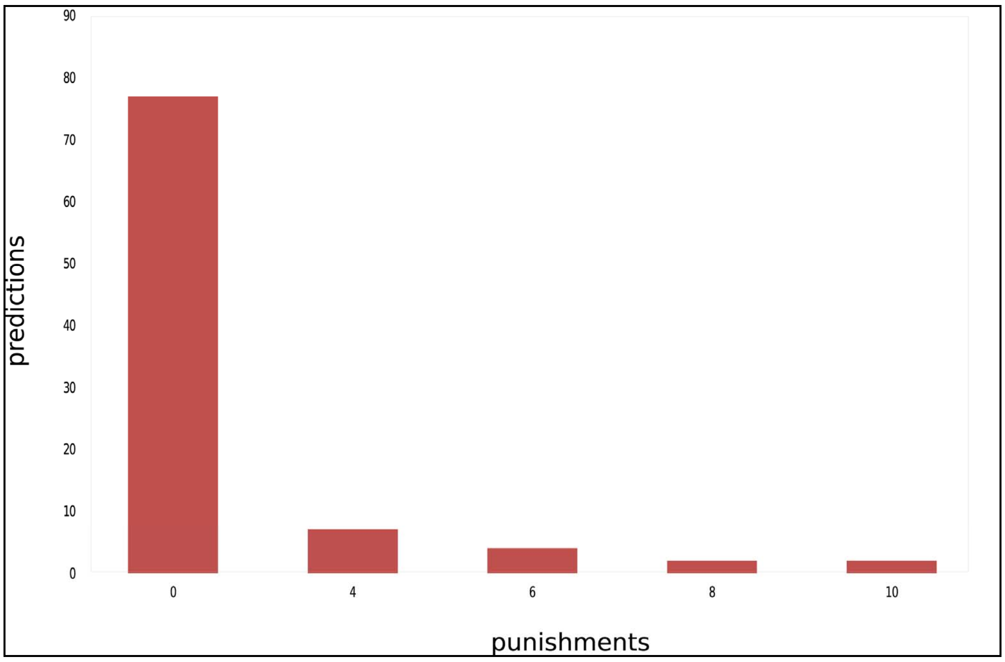

Figure 7. Results from Jaxen on package level.

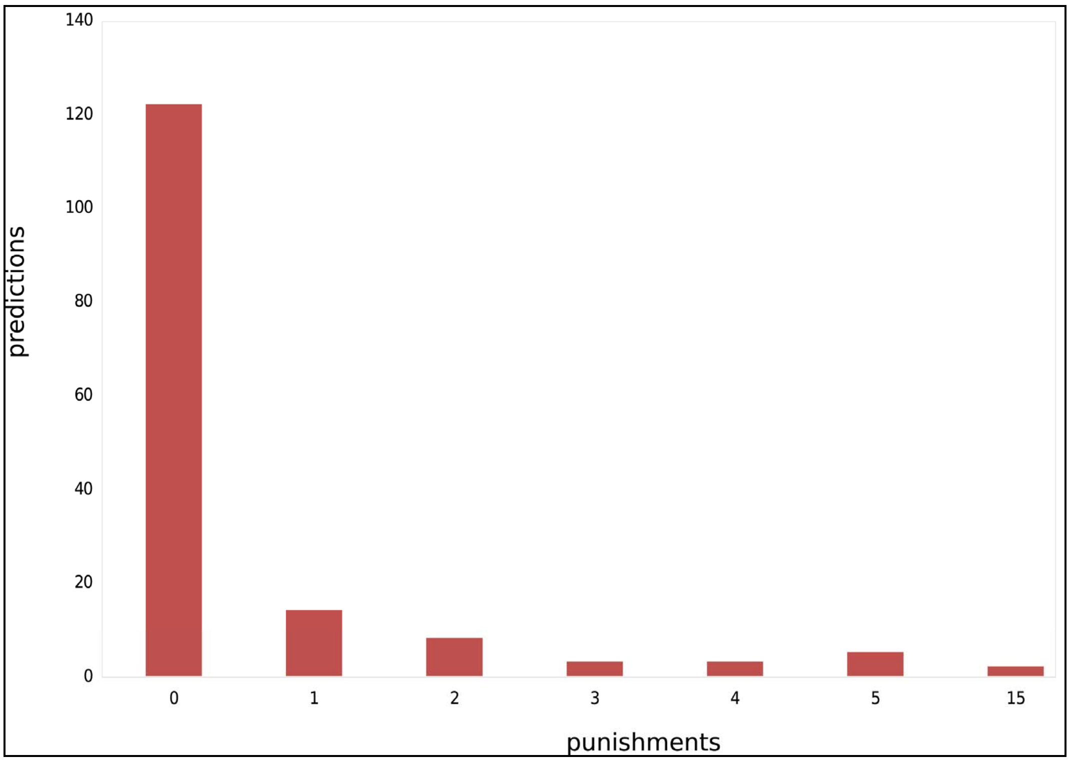

Figure 8. Results from Jaxen on class level.

us that practically there isn't interaction between packages, which characterizes the project as self-contained, and even on class level, the interaction inter classes is low when compared to JodaTime. These findings suggest that our approach is a promising technique for both program understanding and predicting impacts.

As for future work, an improvement for IMPEX would be to automate the process of validating the prediction model based on the two thirds of a repository with the "future" revisions (last one third of the repository). Right now, IMPEX produces automatically the prediction model and predictions (differences in the source code and execution behavior between two revisions from the last one third of the repository), but the calculation of the impact punishments is done manually. Also, integrating the values from the prediction model into a development IDE, such as Eclipse, would be ideal; so while the developer is changing a certain part of her code, she would be able to see it in the development environment, where would her change impact.

As for the approach presented in this paper, it would be interesting to investigate whether there is a relation between the source code changes and the generation of bugs in the software projects.

REFERENCES

[1] D. Schuler, V. Dallmeier and A. Zeller, "Efficient Mutation Testing by Checking Invariant Violations," In: *Proceedings of the* 18*th International Symposium on Software Testing and Analysis*, ACM, 2009, pp. 69-80. doi:10.1145/1572272.1572282

[2] B. J. Grun, D. Schuler and A. Zeller, "The Impact of Equivalent Mutants," *International Conference on Software Testing*, *Verification and Validation Workshops*, Denver, 1-4 April 2009, pp. 192-199.

[3] M. Kim, D. Notkin and D. Grossman, "Automatic Inference of Structural Changes for Matching across Program Versions," *International Conference on Software Engineering*: *Proceedings of the* 29*th International Conference on Software Engineering*, Vol. 20, No. 26, 2007, pp. 333-343.

[4] JodaTime—Java Date and Time api. 2012. http://jodatime.sourceforge.net/.

[5] Jaxen: Universal Java Xpath Engine. 2012. http://jaxen.codehaus.org/