**IMPACTED CLASSES**

The Impacted Classes(IC) approach [x] boosts the SPL **evolution** analysis. It contains further optimizations based on behavioral preservation properties [x] that reduce the performance cost of checking SPL refinements. In some evolution scenarios, when developers change the assets, we only need to ensure that the transformed assets refine the original ones.

Instead of checking a huge amount of products, this technique focuses on testing only the **impacted** assets. Thereof, it avoids generating and testing all product configurations which contain modified classes. It leads to a major reduction on time compared to other proposed techniques since we do not evaluate whole products, like others techniques do. Therefore, the evaluation using Impacted Classes tends to be faster.

To exemplify how the Impacted Class technique works, the figure 0 illustrates a software product line evolution scenario. On the far left side of the figure we have the classes before (source version) and after (target version) the evolution. These classes are submitted to the AST comparator component, who is responsible for identifying the modified assets using static analysis. For each modified asset, the tool computes its dependences using data-flow analysis, that is, the set of other assets needed to compile the modified asset. We call these set of modified assets with their dependences as sub products.

Finally, the approach compiles the source and target versions of each sub product and then uses Safe Refactor to check, for each sub product, whether they have compatible observable behavior, generating test only for modified classes. Safe Refactor is a tool for checking behavioral changes. First, it checks for compilation errors in the resulting program, and reports those errors; if no errors are found, it analyzes the results and generates a number of tests suited for detecting behavioral changes.

Safe Refactor identifies the methods with matching signature (methods with exactly the same modifier, return type, qualified name, parameter types and exceptions thrown) before and after the transformation. Next, it applies Randoop, a Java unit test generator, to produce a test suite for those methods. Randoop randomly generates tests for a set of methods given a time limit. Finally, it runs the tests before and after the transformation, and evaluates the results. If the outcomes are divergent, the tool reports a behavioral change, and displays the set of unsuccessful tests.

Note that, this approach approximates the product line refinement checking since the problem of program verification is undecidable. This strategy can only gain a limited amount of knowledge about a program's behavior by reasoning about certain aspects of the program. When no errors are found we gain confidence in the conformance of the implementation to the specification, but errors may remain. In order to check product refinements, we could use formal proofs, however it could be expensive since they are time consuming and only experienced and expert people can do them. So, proofs are usually done for critical systems (or for some critical operations of a system) only. We lose precision, but it avoids the need for formal proofs that are not accessible to developers. This is what Safe Refactor does.



Figure 0

To better understand how the technique computes the modified class dependencies, consider the figure 1 where the *TestSuite* class extends the *TestCase* class and has a field of type *TextualStep*. For this class, the tool generates the sub product consisting on the following set of classes: {*TestSuite.java*, *TestCase.java*, and *TextualStep.java*}

After identifying the changed assets using the AST Comparator, it uses the Soot framework for computing their dependences.

* Explain how the dependencies are computed?
* Cite Soot Framework and Data-Flow Analysis.
* Do not forget to mention the forwards flow analysis.
* The tool recursively computes dependences for each identified dependence

As we can see, this tool only checks the modified classes and does not generate all

products impacted by the change, optimizing the evaluation.

After identifying the changed assets using the AST Comparator, it uses the Soot framework for computing their dependences. It implements a data-flow analysis. The union of the changed asset and its dependences generates a sub product that is a minimal set of classes that can be compiled to perform tests that Safe Refactor uses to evaluate them.

Figure 0 shows the modified assets and their respective dependences identified by the tool.



Figure 1

Impacted Classes is suitable only when feature model and configuration knowledge do not change or when they are at least refined.



Figure 2

This figure illustrates an SPL refactoring, where the developer changes the *TestSuite* class without modifying the Configuration Knowledge and the Feature Model. It also shows a refactoring in a method (*isEquals*) responsible for comparing two objects.As you can see, before the evolution we have an if-else branch that firstly performs an *exclusive* *or*, which is evaluated to true, if one, and only one, of the input objects is equals to null. Following the method logic, the else branch means that we have two possibilities: either both objects are null or both objects are not null. Therefore, in the case of the *if branch* is evaluated as false, on the else branch, it is only needed to check if the first object is different to null, which if evaluated as true, implies that the second object is also not null and the comparison can be made.

Following the same reasoning, if the first object is null, even the second object being concrete the comparison could not be executed since it would throw a NullPointerException.

On the evolved method, the developer removes the *xor* verification and only checks if the second parameter is equals to null, since the first object is already checked on the upper method *compare,* as you can see in the *TestSuite* class on the far right side of the figure 2.

Now that you already know what this evolution is about, let´s describe what the technique does in this scenario. For this evolution as explained before, Safe Refactor generates and run unit tests for the *TestSuite* methods and then apply these test before and after the evolution to compare the behavior of the sub product. In this case, *randoop* creates a test where the first parameter is null and the second one is a concrete object. When Safe Refactor executes this test case on the original class, nothing abnormal happens and the method executes flawlessly. However, we cannot say the same for the target class, since it throws a NullPointerException while trying to execute the method *isEquals*. Therefore, the tool reports a behavioral change and shows the set of unsuccessful tests.

Note that, Impacted Classes performs a narrow minded analysis, too focused on the impacted method. It totally approximates the evolution checking, since it only analyses the impacted class during the SPL maintenance, ignoring the whole context where the change has been applied and hastily consider the SPL evolution as a Non-Refinement. Nonetheless, as the method *compare* is the only one called by the backwards impacted classes and it has compatible observable behavior after the evolution, we assume that the approach produced a false-negative and the SPL is a refinement considering the final user perspective.

Our study confirms that *IC* presents some false-negatives, and it especially happen when during software maintenance, the developer removes requirements previously implemented in superior classes or when a local change, which produces faults to itself without spreading out this behavioral change to the upper method call hierarchy. Or even when the developer is oriented by Test Driven Development and creates a unit test, which passes aspecific parameter exposes a method fault, however this argument will never be passed by the most external layers of the system.

Therefore, we may lose precision with Impacted Classes, since local changes in OO classes may indirectly impact other ones, and this tool only focuses on changed classes without taking into account all the contexts where it is used in the product line.

We suppose this evolution is applied to improve readability and to reduce redundancy of object state verification since the first object is already checked in upper classes, is only necessary to check if the second object is different to null, and if it evaluated to true, the comparison is made. However, if we focus only in the local change and create tests for it, we will see that if the first object parameter is null, the method throws a null pointer exception since the new, evolved method only verifies the second parameter. On the other hand, if we expand our analysis to the whole SPL, we will see that this local behavioral change do not compromise the SPL functionalities or in order words do not negatively impact the SPL products clients.

In order to solve this problem, we propose an approach called extended impacted class, which goes beyond of focus on changed classes without taking into account all the contexts.

In Figure 3 we show how the Extended Impacted Classes overcome this problem.

* Explain How Extend Impacted Classes Works.
* How we find the backward impacted classes.
* How we know these classes are below the Graphical User Interface.
* Provide an Algorithm

As you can see, instead of checking the modified class, we are interested in

The idea is to capture dependencies between the class a programmer is maintaining and the others.

We might not impact with our current maintenance task.

The technique then become aware of the dependencies and, consequently, might avoid the false-negatives problems described in the aforementioned scenarios.



Figure 3

As you can see, instead of checking the impacted class, we are interest on the backward impacted classes.



The tool first checks, without generating products, if the target SPL is well-formed

(Step 1), which means that it generates well-‘

Step 2 generates product configurations and maps source products to their likely corresponding products when they exist. We use the Alloy Analyzer for generating the product configurations from the source FM. Then, we construct source and target sets of assets using the source and target FM and CK.

Then, we bypass asset and product refinement checking and just evaluate the

CK for every possible configuration present in FM, checking if all existing evaluations of CK with the configurations of FM are still present in the evaluations of the resulting CK and FM. In this case, the target FM and CK jointly refine the source FM and CK (Step 2) [Bor11].

If this condition is not satisfied, we cannot use this technique.

We can only use … All product pairs could be applied. Cite app and succinctly describe it!

If they do not find changes, they assume that the line is refined;



Figure 6

Figure 6 shows the approach variation. On the left side, we have the technique who considers only one level above the impacted class and on the right side we have the strategy, which considers several levels above the change. It walks up through the superior hierarchy of classes to reach the closest level to the interface as possible. The former is expected to be easier to evaluate, since it does not have to generate test for structurally complex objects, however, we do not know in which level the system camouflages a negative change such as a conditional structure makes a piece of code unreachable.

The second approach faces the complex object issue aforementioned, but on the other hand, it overcomes the masked-changed problem cited as well, which we are most interested in. A second reason why we prefer to adopt the second approach is because we can integrate new testing tools specifically designed to efficiently generate structurally complex inputs with thousands of objects. As an example, we could use Shekoosh, a novel framework for generating large data structures. Given a Java predicate that represents the desired structural and data integrity constraints, and the size of the structure to be generated, the Shekoosh test generation algorithm produces a structure that has the given size and satisfies all the constraints.

**EVOSUITE**

EIC decreases false-negatives, but on the other hand, it produces more false-positives and our study concluded that the main reason is the limitations of Feedback-Directed Random Testing with Randoop. Specially, because *Randoop* does not directly aim at code coverage, and unfortunately, achieving higher coverage leads to higher probability of finding faults. Its only goal is to find test sequences for which the system under test fails. It does so by executing random sequences of methods from the class-under-test. Since the sequences are random, results may vary slightly when examining results from different runs.

Using prior results; more intelligent inputs and sequences will be chosen to avoid redundant or illegal sequences as much as possible, making up the feedback-directed aspect of the tool. For instance, a sequence containing code that will result in an exception is pruned from future runs [1]. The tests that are output can be classified into two categories: first regression tests that capture the current behavior of the code are generated, and secondly, also contract violating tests are produced, which could indicate potentially buggy code. This brings to mind another Randoop’s restriction; the set of contracts that Randoop verifies is relatively small; its designers wanted to be conservative and only check for contracts whole violation is highly likely to indicate an error. The tool should let the user specify more contracts to check by writing a special "contract-checking" class.

Randoop's main attraction is the automated generation of a full regression test suite. This can be helpful when working on a project that has no tests whatsoever and for which it is crucial to ensure current behavior is preserved when updating the code. Even though one gets free and effortless regression tests, it comes at an excessively high cost in maintenance. When running Randoop for 30 seconds, it yields some 96,000 lines of testing code, accounting for more than 400 test cases. Some of these cases use a hundred variables that are named by the unhelpful convention *var\_Number*, making any deep understanding of the tests practically impossible.

Originally, we are not concerned about test readability, nevertheless, when this prototype become an eclipse plugin, the developer might maintain and evolve the test suite, what makes test legibility a must requirement to enforce.

To overcome the abovementioned problem, we integrated evosuite to our toolkit. A novel paradigm in which whole test suites are evolved with the aim of covering all coverage goals at the same time while keeping the total size as small as possible [x]. Evosuite uses an evolutionary technique in which, instead of evolving each test case individually, it evolves all the test cases in a test suite at the same time.

Since coverage goals are not independent, not equally difficult, and sometimes infeasible—the result of test generation is therefore dependent on the order of coverage goals and how many of them are feasible. As a result, this strategy prevents waste of time in unreachable branches. This fact makes evosuite outperform randoop at code coverage, because the latter does not even consider it as a criteria on the search for tests.

In addition, *Evosuite* includes the total length of a test suite as a secondary optimization goal. As the stopping conditions are based on coverage achievement, *evosuite* therefore minimizes test suites as a post-processing step. This feature enhances test readability and performance, because the test suite is smaller and accordingly the techniques spend less time to compile and run tests.

Below, we show how details how evosuite works.

The author of this tool provides a more detailed and technical explanation of its workings in Sect. 2 of [3].

Coverage measurements using Emma were conducted to get a quick overview of the quality of the generated regression tests.

* Why Randoop contributes to the approach false-positives?
* What its limitation?
* What did you do to solve this problem? Evosuite
* How evosuite works?
* Why this is a great solution?
* **Show details about evosuite evolutionary strategy**

**References**

[1] [BTG10] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. A theory of software product line refinement. In ICTAC, pages 15–43, 2010.

[2] C. Pacheco and M. D. Ernst, \Randoop: feedback-directed random testing for Java," in OOPSLA 2007 Companion, (Montreal, Canada), ACM, 2007.

[3] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, Feedback-directed random test generation," in ICSE '07: Proceedings of the 29th International Conference on Software