**IMPACTED CLASSES**

The Impacted Classes approach [x] boosts the SPL **evolution** analysis. It contains further optimizations based on behavioral preservation properties 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, the evaluation using Impacted Classes tends to be faster.

Figure 1 depicts the Impacted Classes technique. First, the tool identifies modified assets using syntactic 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 this set of modified assets with their dependences as sub products. The approach compiles the source and target versions of each sub product. It then checks, for each sub product, whether they have compatible observable behavior, generating test only for modified classes.

This refactoring changes the assets without modifying the CK and the FM.

Figure 2 shows the modified assets and their respective dependences identified by the tool. The rounded rectangle illustrates the sub product, which contains the impacted class and its dependencies.



Figure 1

The *TestSuite* class extends the *TestCase* class and has a field of type *TextualStep*.

the tool computes dependences for each identified dependence

For this class, the tool generates the sub product consisting on the following set of classes: {TestSuite.java, TestCase.java, and TextualStep.java}

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

products impacted by the change, optimizing the evaluation. However, it is important to stress that although costly, we can use All product pairs to check any kind of evolution scenarios, while Impacted Products and Impacted Classes are suitable only when feature model and configuration knowledge do not change or when they are at least refined.



Figure 2

Impacted Classes is suitable only when feature model and configuration knowledge do not change or when they are at least refined. **Moreover, with Impacted Classes we may lose precision, 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.**



Figure X

The figure x shows a method refactoring during a class evolution. The developer breaks the method *one* behavior when making it return 1 instead of 0. The method *two* accesses the impacted method, nonetheless it has the same behavior since itreturns 10 regardless method *one* returns 0 or 1. This approach ignores the context where the impacted class services are used and how they are accessed; thus it wrongly states that the evolution is not a refinement. As the method *two* 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 by the final user perspective is a refinement.

Masked-change Definition:

A refactoring that change a piece of code, breaks its behavior, however this change is not perceived by the upper classes.

A camouflaged change by the context

A class local change, which might produce failures to itself without spreading out this behavioral change to the upper method call hierarchy

When you remove requirements previously implemented in superior classes

When a specific parameter exposes a method fault or error

A class local change, which might produce faults to itself without spreading out this behavioral change to the upper method call hierarchy

A specific parameter exposes a method fault, however this argument will never be passed by the most external layers of the system.



This figure shows a refactoring in a method responsible for comparing two objects.

The developer removes in a most internal class a verification of a null parameter, which is previously checked in an outmost class.

As you can see, before the evolution we have an if-else branch where firstly we a xor verification of both objects, if at least one object is null, the method early returns false, otherwise we check if the first object is null because the

The *if* branch is evaluated to true, if one, and only one, of the input objects is equals to null, thus the else branch means that, we have two possibilities: both objects are null or both objects are not null. Therefore, it is only needed to check if the first object is different to null, which if evaluated to true, implies that the second object is also not null and the comparison can be made.

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.

What is the solution for this problem?

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**

precipitated/lazy fault detection



Explain Extended Impacted Classes

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

(Step 1), which means that it generates well-formed products, that correspond to valid products in the underlying languages used to describe assets [1]. If it finds a problem, it stops the process, reports all the invalid product configurations found and indicates that the product line is not refined and the evolution is not safe.

If it does not find a problem …

We reuse Felipe’s implementation of an Abstract Syntax Tree (AST) to compare each version of the code assets and identify the set of impacted classes during code maintenance.

It identifies the backward impacted assets using syntactic analysis.

Soot [2] is a framework for analyzing and transforming Java programs. Soot uses intermediate representations of programs, with the most prominent being Jimple, a typed 3-address representation designed for optimizations.

Soot is a framework—written in Java—for representing, analyzing, and transforming Java class files[5]. The framework is designed to facilitate research into the optimization of Java bytecode,

**PROVIDE ALGORITHM AND MOTIVATING EXAMPLE**

**References**

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