Figure 0

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 results 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.