Abstract

A software system is increasingly more difficult to evolve over time unless appropriate measures are taken. The three examples below show some technological support for checking specific constraints during the development of the next release in the hope that the resultant release will be of a higher quality than without such checking. The first example is that of re-engineering evolution-prone modules, the second example is that of ensuring consistency between code and documentation, and the third example is that of verifying regression test coverage.

Example 1: Re-engineering evolution-prone modules

Mattsson and Bosch\(^1\) have proposed an approach to identify those modules of a system that require re-engineering. Proactively maintaining the software (an object-oriented framework in their case) by restructuring the evolution-prone modules could "simplify the incorporation of future requirements"\(^1\). In their approach, the evolution-prone modules from past releases are identified based on their size, change rate and growth rate.

Once it has been decided which modules or components have to be re-engineered during the development of a particular version, one issue is to ensure that all of the identified modules do go through such an effort. Such verification can be done on a process model, using the following policy:

\[
\text{Policy: } \forall c \in \{ p \in \text{TypedEntSet("Component") } | \text{ p.name } \in \text{<list of components>} \} \bullet \exists r(a,m,t) \in \text{TypedRelSet("activity consumes component") } \bullet a.\text{name} = "\text{re-engineering}"
\]

in which the part <list of components> has to be replaced by the actual list. For example, in the case of the V-elicit system used in Section 2.1, one such list could be {"view_matching_V1","generator_V2"}. The meaning of this policy is that for each component in the list provided, there should be an activity “re-engineering” consuming it.

A process model can then be checked against this policy. The model can either represent the planned process before its enactment (with the policy checking mechanism used to ensure the plan is complete), or represent the process as it has been performed up to the time when the model was elicited (the policy checking acting as a process monitoring tool in this case).

As an example, Figure 1 shows a process model representing the planned process for the development of the sixth version of the V-elicit system. For simplicity, it contains only the high-level activities to be performed:

"makeChanges", "re-engineering", and "testing"). More specifically, it identifies the "view_matching" component will need to be re-engineered (moving from version 1 to 2). As one can see here, this plan is not correct: the "generator" component (version 2 -- which was identified above for re-engineering) is mistakenly left out from the re-engineering effort (i.e., this component is not an input to the "re-engineering" activity box in Figure 1). Such mistakes do happen when building prescriptive models in the planning phase, even in moderate sized projects. This is why it is quite important to verify the planned process (prescriptive model) - against the prescribed policies -- prior to its execution, in order to prevent development (and hence evolution) errors.

Figure 2 shows the result of verifying the plan against the described policy results in the listing of violations - that is, those components that were supposed to be re-engineered but have not been included in the plan. Note that, in essence, this is an automated plan inspection, which is much preferable to hand-checking the plans as carried out (if at that!) in practice today. Its value is particularly felt in large or complex systems; when many individuals are involved in the project; when quality is at stake; and when time is at a premium.

It should be noted that this policy could easily be modified to detect automatically which components should be re-engineered. For example, one may want to enforce that components having a change rate higher than a certain level should be re-engineered after a given period of time. Such an assumption can be formally defined and incorporated in the policy above, replacing the fixed list of components.

![Figure 1 - Overall process model for the development of the sixth version of V-elicit](image)
Example 2: Ensuring consistency between code and documentation

One problem in evolving software is to keep an accurate documentation of the system. As noticed by Parnas\textsuperscript{2}, "when they [maintainers] document their work, it is often by means of a memo that is not integrated into the previously existing documentation". Then, the documentation is not used because it is no longer accurate. However, as shown in a case study performed by Tryggeseth\textsuperscript{3}, the availability of documentation during maintenance increases system understandability and maintainers productivity.

One approach in solving this problem has been to re-document a system (using for example reverse engineering techniques when the documentation is no longer useful). But a preventive approach would be to ensure that at the end of each new-version development, the documentation still reflects the implementation. As an example, one may want to check that the code implements exactly the class diagram in the design documentation (i.e., no classes missing or added, and all attributes and methods properly implemented). This could be done by comparing the class diagram from the design documentation with a class diagram generated by reverse-engineering the code. The two diagrams thus obtained could be merged into one model, and be checked by a set of policies representing what should be similar in the two diagrams. As an example, the following policy verifies that all classes in the code are included in the design documentation.

\[
\text{Policy: } \forall c_1 \in \{ c \in \text{TypedEntSet(\text{Class})} \mid c.\text{source} = \text{“code”} \}
\]

\[ \exists c_{2} \in \{ d \in \text{TypedEntSet("Class")} \mid \text{d.source = "documentation" } \} \bullet c_{1}.\text{name} = c_{2}.\text{name} \]

Other policies could be written to check that the related classes have the same attributes, and the same functions (having also the same parameters and return value).

As an example, Figure 4 shows the result of verifying the diagrams in Figure 3 against this policy.

Boxes in red are the classes identified in the design documentation, and the boxes in blue represent the actual code structure (classes implemented).

**Figure 3 – Simple class diagram from design documentation and actual code structured as re-engineered**
Example 3: Verifying regression test coverage

This last example focuses on testing. Specifically, a desirable outcome of a software modification is that the new release satisfactorily implements the retained features from the older release. However, injudicious use of regression tests can excessively (or insufficiently) exercise software components that have undergone little (or significant) change, reducing the effectiveness of the tests in terms of the number of defects found per effort spent in running these tests. Munson has thus proposed that testing effort be proportional to the "code churn" a module has received (i.e., amount of change across releases -- in terms of complexity). Thus, assuming that modules have the attributes "code_churn" and "actual_profile" (i.e., proportion of the testing activity applied to each module), then the following policy could verify proportional testing:

\[
\text{Policy: } \forall m \in \text{TypedEntSet("module")}\left|\frac{m.\text{code\_churn}}{\sum_{n \in \text{Module}} n.\text{code\_churn}} - m.\text{actual\_profile}\right| \leq 0.1
\]

That is, for each module \( m \), the difference (in absolute value) between its relative code churn (i.e., the module's code churn divided by the total code churn for all modules) and its actual profile should be no more than 0.1 (i.e., 10 percent of maximum difference).

\(^4\) Personal communication.
Figure 5 shows that checking test data from Munson's example against this policy identifies modules A, B, D, K and Q (from a set of 36 modules) as violating the “proportional testing” policy. The implication of this finding is that some modules are under-tested, resulting in possible defect slippage, and its consequential runtime failures and risks to the society, and increased evolutionary costs; whereas, some others are over-tested, leading to unnecessary delays and increased cost of quality.

The stated policy is the same as that manually identified by Munson, with the added advantage of automation – especially in large production systems. It is then a management issue as to when such automated checking should be conducted, e.g., after regression testing (as depicted in Figure 5), or repeatedly during testing to optimise coverage, or preventatively during test planning.