Software Product Line Engineering

Software Product Line engineering, a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.[ <http://www.sei.cmu.edu/productlines/frame_report/what.is.a.PL.htm>]. However, a scalable modelling and an efficient method to verify the system behavior needs to be put forward to meet the requirements. A featured transition system(FTS) has been purposed to obtain the behavior of each product of the Software product line. FTS is able to reason about the whole product line, or subsets of it, and model very detailed behavioral variations. Moreover, it could run model checking tool freely and take account into the feature dependencies and incompatibilities. Another model checking technique has been introduced to verify LTL properties for all the products of an SPL at once and would point out the products that violate the properties.  linear-time temporal logic is an infinite sequence of states where each point in time has a unique successor, based on a linear-time perspective. [http://www.cs.colostate.edu/~france/CS614/Slides/Ch5-Summary.pdf]

Using SPL could help to take economic advantage of the fact that many of the products are very similar as planned and deliberate, strategic decisions are made to impact decisions systematically.

Model-checking technology - Product-Line Verification:

Model-checking technology is used to exhaustively and automatically check whether this model meets a given specification. Typically, one has hardware or software systems in mind, whereas the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Model checking is a technique for automatically verifying correctness properties of finite-state systems and specifically product line verification is one type of model checking technology. In product line verification, there are three strategies which are product-based, sample-based and family based strategy. However, before [], there is no comparison between them in a controlled setting, therefore, [] provides case studies and experiments on that. And they have created an analytical model to describe the trade-offs of the individual verification strategies and given promising results after revisiting the discussion of the strengths and weaknesses of sample-based and family-based strategies, shown in figure, figure, figure. Furthermore, a generic conclusion is given that the success of a sample-based strategy depends on the defect and sample rates, whereas the success of the family-based strategy depends on the similarity between products. Also, sampling can save verification time but could lead to miss the defective products.

Scalable Analysis of Variable Software

Variability Analysis, known as family-based analysis, is not to generate and analysis variants separately but to directly analyze the variable code base by utilizing some configuration knowledge. Although, there are several proposals on this theme, there are no real-life, large-scale systems being implemented due to its potential challenges. It requires more effort than traditional analysis of a single system as all local variants are considered into the analysis. But the effort is worth taken as it could avoid analyzing the common code by taking advantage of similarities among variants. Therefore, [] proposed a way to detect whether variability-aware analysis scales to large systems, as it should consider all code and all variations of a system simultaneously. Before [], they have already come out with a sort of Java libraries which could handle the scaling in Java written project such as Featherweight Java, Lightweight Java, the lambda calculus, and other dialects of Java. They also took advantage of other researchers' variability-aware approaches for data-flow analysis using liveness analysis, which is a traditional data-flow analysis to compute whether variables are live, which may be read before being written again for a given statement. Its result can be used to conservatively detect dead code, useless code. A practical, scalable, variability-aware and sampling-based analysis for real-world, large-scale systems written in C is finally purposed. Experiments on large-scale system including Linux Kernel is taken and the performance of variability-aware analysis outperforms some of the sampling heuristics and still being complete and some of the limitations on pair-wise, sampling heuristics are found. Also some future work regarding to other large system should also be included to test the achieved results.