Software Product Line Implementation Techniques: Bachelor/Master Thesis Topics

Software product lines and other highly configurable systems represent not just a single software product, but a whole family of software products. By choosing a suitable configuration, a highly configurable system can be customized to meet a specific set of requirements. An example is the Linux kernel, which has more than 12,000 configuration options. These options allow the creation of specialized Linux kernels suitable for anything from wireless routers to high performance computing clusters.

Metrics for Variability-Aware Code Smells

Much of the configurability of highly configurable systems depends on variable code structures (for instance, conditional compilation a.k.a. ifdef). However, this variability introduces another dimension of complexity into the code, which, in turn, creates new possibilities to make bad design decisions.

Code smells are an established concept to describe badly-designed code in a software system. Although smelly code is not buggy code, code smells do impede program comprehension, maintenance and evolution. Unless corrected, code smells decrease programmer productivity and make incorrect modifications, that is, the introduction of bugs, more likely.

However, existing code smells do not take variability into account. Therefore, they are unsuitable to describe certain code smells that can arise in software product lines. In an ongoing collaboration with Sandro Schulze from the TU Braunschweig, we are working on a catalog of variability-aware code smells, that is, code smells that explicitly take variability into account. A number of smells have already been developed. Although we have found real-world systems that exhibit these smells, there is currently no empirical evidence on how wide-spread these smells are in practice. The goal of this thesis is to develop a set of metrics to automatically detect these smells. These metrics should be implemented in a tool. This tool could be based on existing analysis tools for C-preprocessor-based product lines or feature-oriented product lines.

Qualitative Analysis of C Preprocessor Usage in Software Product Lines

The C preprocessor is a wide-spread tool for the implementation of software product lines. However, the C preprocessor is used for many purposes besides variability, such as

- defining constants and abbreviations, and realizing ad-hoc inlining (via #define)
- reusing function and type declarations (via #include),
- overcoming language limitations,
- commenting out blocks of code (#if 0 ... #endif).

A number of studies have looked into C preprocessor usage in traditional, single software system engineering (for instance, Ernst and others, TSE 2002). Other work has analyzed C preprocessor usage specifically in the context of software product lines by applying a set of metrics (Liebig and others, ICSE 2010). However, we have little detail on how much C preprocessor usage is related to variability and how much is related to other purposes, such as the ones above. Furthermore, we have little information on why portions of code are variable. One reason for variability is, of course, the implementation of features that are relevant to end users, such as support for UTF-8 strings in a text editor. However, there may be other reasons, such as:

- usage of non-standard functions (for instance, iconv has differing signatures on Linux and FreeBSD),
- platform-dependent header locations,
- workarounds for compiler bugs.
The goal of this thesis is to gain a better understanding of how and why the C preprocessor is used for the realization of variability in annotation-based software product lines. The result could be classification or a catalog of patterns of variability-related C preprocessor usage.

In future work, the use or disuse of these patterns could be analyzed empirically. With respect to program comprehension or maintenance, some preprocessor usage patterns are probably better than others. Thus, another avenue of research could be to identifying which patterns are recommendable and which are not.

**Understanding Autotools Usage in Software Product Lines**

Thanks to a large body of existing work, the variability model of the Linux kernel and the tools, languages and techniques that implement this model, are well-understood. However, a large number of configurable software packages for Unix-like systems uses the GNU build system (a.k.a. Autotools) to achieve portability and other variability-related goals, which is less well-studied. Therefore, existing product line development tools, such as FeatureIDE, or analysis tools, such as TypeChef, cannot directly be applied to Autotools-based software.

The goal of this thesis is to investigate how Autotools is used for the implementation of software product lines. One aspect of this investigation is Autotools’ support for feature relationships and dependencies, such as optional features, mutually exclusive features, cross-tree constraints. With this knowledge, extracting a feature model from an Autotools system may become feasible. Consistency checking of Autotools configuration files and source code is another challenge and may a valuable contribution to the community. Furthermore, traceability of features from Autotools configuration scripts through build scripts down to source code would be of interest.