Reflections on Feature Oriented Software Engineering

Paul Klint (paulk@cwi.nl) & Tijs van der Storm (storm@cwi.nl)
Centrum voor Wiskunde en Informatica (CWI)
Kruislaan 413, NL-1098 SJ, Amsterdam

August 27, 2004

Introduction

Feature Oriented Software Engineering is an emerging discipline in which feature models [2] are used to model the commonality and variability in a product family. It is our that view product family engineering of very large and highly variable systems cannot do without adequate tool support. When the number of variants that have to be delivered and maintained is exponential, automatization is not only desirable, but inevitable.

In this position paper we address two problems that stand in the way of wide adoption of feature oriented software engineering:

- Feature models don't scale for application to very large systems.
- The "white-board distance" between feature models and source code is too large.

The rest of this paper is organized as follows. First we present solutions to the problem that feature models with exponentially many variants are hard to prove consistent. Additionally we present work in progress on an interactive environment for feature models. Both can help in scaling feature models for highly variable systems. The second part addresses traceability issues in an very broad context. We present areas of future work by discussing the relation between feature models and source code.

Scaleability

For feature models of highly variable systems, we identify two problems of scaleability:

- Proving a feature model consistent by enumerating all variants is infeasible (the configuration space is exponential).
- Maintenance of large feature models is hard (especially if they are specified graphically and informally).

This section presents tools and techniques implementing solutions to these problems. All tools operate on *feature descriptions* [6]. Feature descriptions are a textual analog of feature diagrams, and are therefore more amenable to formal analysis and transformation. Moreover, they are much more compact than graphical diagrams.

Binary Decision Diagrams

We have tackled the first problem by transforming feature descriptions to Binary Decision Diagrams (BDDs) [1, 5]. BDDs originate from modelchecking and can be used to efficiently represent boolean functions.

The consistency problem of a feature description can be cast as a *satisfiability* problem for boolean sentences. Satisfiability is obtained by transforming a proposition to a BDD. A BDD is an if-then-else tree, with logical variables (atomic features) as conditions. If common subtrees are shared, one obtains a directed acyclic graph. A boolean sentence is satisfiable, *iff* such a graph can be constructed and this graph is different from falsum.

Checking the consistency of a selection of features is equivalent to the *satisfaction* problem for boolean sentences. This amounts to finding a path to the unique *true* leaf in the associated BDD.

Experiments show that the reduction in size obtained by representing the configuration space as BDDs is enormous.

Feature Analysis and Manipulation Environment

For maintenance of large feature models we propose an interactive Feature Analysis and Manipulation Environment (FAME). FAME provides explicit support for evolving and refactoring feature descriptions.

Modifying large graphical feature models by hand is unpractical and error-prone. Therefore, FAME is

based on textual feature descriptions. FAME supports systematic editing of feature models through evolution operators. These evolution operators range from simple edit steps to semantics preserving refactorings such as fold/unfold or permute arguments.

To inform the user as much as possible about the effect of her modifications, FAME includes a well-formedness checker. For presentation purposes, feature descriptions can be pretty printed, or displayed as a graphical diagram. FAME can also provide a number of statistics about feature descriptions, such as, the number of variants, the number of atomic feature, the maximum feature expression fan out etc.

In addition to evolving and analyzing a feature description, FAME can be used to check the description for inconsistencies with the derived BDD. This BDD is also used to validate feature selections.

Traceability

One of the key challenges for feature oriented software engineering is bridging the gap between problem domain (feature description) and solution domain (implementation). The question is: how can we guarantee that every valid feature selection corresponds to an implementation variant that is also valid (e.g., type-correct, executable, functional etc.)? Since it is likely that feature model and implementation are maintained separately, that is, they *co-evolve*, it is imperative to keep feature model and implementation in sync. For highly variable systems, manual maintenance of this relation is not an option.

The above analysis suggests two major approaches to close the gap between feature descriptions and source code.

In a bottom-up approach, all signs of variability in existing source code are spotted semi-automatically and are used to infer a feature model for that code. Certain design patterns and programming conventions can be used as indicators for variability. The resulting feature model can be used to better understand and manage the variability of existing code. This approach mainly raises research questions aiming at *variation point recognition*.

In a top-down approach, each atomic feature in a given feature model is associated with code fragments that implement that feature. After selecting a specific variant of the description, the code fragments of all selected features as merged into an implementation. This approach is much more ambitious but also raises many more research questions:

- How are code fragments expressed (in a dedicated language or in an extended existing language)?
- How do code fragments interact with their environment?
- How do code fragments interact with each other?
- How are these code fragments type-checked compositionally (i.e. without having to check all valid combinations)?
- How is the generated implementation tested?

Going from aspects [3], via concerns [4], to feature diagrams we observe an increasing level of structure and expressivity at the level of describing variability. This should be matched by increased expressivity at the level of code composition.

Both approaches just sketched and the tools that are needed to support them will be essential to make progress in the emerging discipline of *Feature Oriented Software Engineering*.

References

[1] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[2] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, CMU, Pittsburgh, PA, Nov. 1990.

[3] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[4] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings ICSE'99*, 1999.

[5] T. van der Storm. Variability and component composition. In J. Bosch and C. Krueger, editors, *Software Reuse: Methods, Techniques and Tools: 8th International Conference (ICSR-8)*, volume 3107 of *Lecture Notes in Computer Science*, pages 86–100. Springer, June 2004.

[6] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, March 2002.