

Research Statement

Kacper Bąk

Generative Software Development Lab, University of Waterloo, Canada

October 8, 2010

1 Background

Software product lines (SPLs) are a way of representing families of related, but customizable software systems built from shared components. The premise of SPLs is to automate the process of software construction, thus making it cheaper, less error-prone and faster. Software product lines are often discussed in terms of the *problem space*, the *solution space*, and *mappings* between them. The problem space specifies products available in a SPL; the solution space describes product-line architecture; and finally, mappings establish dependencies between the two spaces.

Feature modeling is a popular technique for capturing commonalities and variabilities in software product lines. Feature models live in the problem space. The solution space is often viewed as a software architecture described in terms of components and connectors. Elements of the two spaces can be additionally constrained, so that presence of an element requires or excludes the presence of another element. There are several ideas on how to express mappings such as logical constraints, annotations in the solution space (presence conditions), or preprocessor directives (e.g. `IFDEF` in C).

In practice, programmers often use project-specific *ad-hoc* methods to construct software product lines – a well-known example is the Linux kernel. The lack of a systematic approach makes it hard to reuse existing SPLs in different projects. It also complicates reasoning and performing non-trivial analyses such as verifying consistency, checking element liveness, performing configuration completion, and reasoning on model edits. Reasoning helps to uncover errors in SPL instances, remove unused code, and support the configuration process.

My research focuses on finding a systematic way for constructing software product lines and reasoning on them.

2 Current Research

Over the last year I have been working on *Clafer*, a meta-modeling language with first-class support for feature modeling. It provides a concise notation for meta-models, feature models, mixtures of meta- and feature models (such as components with options), and models that couple feature models and meta-models via constraints (such as mapping feature configurations to component configurations or model templates).

In contrast with UML or *cardinality-based* feature models, Clafer provides uniform syntax and semantics for feature and meta-models, thus naturally expressing a wide range of concepts. These concepts span over the problem and the solution spaces and are linked by mappings expressed as constraints. Providing a single notation makes it possible to reason about various models using the same infrastructure. This as an advantage, since end users do not have to bother about manual integration of different reasoning engines or tools.

Clafer has been developed simultaneously with a Clafer-to-Alloy translator. Alloy is a class modeling language with an elegant constraint notation. The translator gives Clafer precise translational semantics

and enables model analyses using the Alloy Analyzer. Different strategies are applied for distinct model classes in Clafer. All of the strategies preserve the meaning of the models, but optimize the analysis runtime by exploiting Alloy constructions. The Clafer-to-Alloy translator is written in Haskell and comprises of several chained modules: lexer, layout resolver, parser, desugarer, semantic analyzer, and code generator. Although it is still a prototype tool, it can handle models with thousands of elements and constraints.

Recently, we evaluated Clafer on over 60 realistic feature models, meta-models, and feature-based templates (FBMTs). A FBMT consists of a feature model, a meta-model, a template, and a set of mapping constraints. A template specifies fixed and variable parts of the architectural structure. Points of variability are eliminated by selecting features from feature model. We performed representative analyses on the above models, i.e. instance finding, checking consistency, and element liveness analysis. All these analyses are theoretically NP-hard problems, but in many cases they can be done in a reasonable time. Interestingly, analyses do not have to be performed on the whole model. Instead, we manually created model slices and checked their properties. We see a great potential in the model-slicing approach and our results were very promising. We obtained acceptable timings (within seconds) for slices of realistic models, while not fully optimizing for the capabilities of the Alloy Analyzer.

We envision Clafer as a pivot language where different models can be mapped into the single notation of Clafer. Next, an automatically chosen reasoner can perform model analysis. In a sense, Clafer has the potential of bringing together software engineering and formal methods. Through the work on Clafer we precisely characterized relationships between feature and class modeling and built a uniform framework to reason about various kinds of models.

3 Research Plans

The fascinating world of software product lines is gaining more attention not only from academia, but also from industry. I would like to contribute to this area by working on efficient analysis of models and integrating analysis tools with programming environments so that they can be easily used by SPL practitioners.

I have started combining Clafer with a programming workbench. Such a combination will effectively embed models into program's code. It will simplify model extraction and remove redundancy that is currently caused by overlapping parts of the code and the model. I strongly believe that making code and model a single entity will solve certain problems with synchronization and will lead to faster adoption of model-based software engineering. Tighter integration will allow humans to reason about software more easily, because it will abstract away program's details.

My long-term goals include work on efficient model analyses and making them available in Clafer. First of all, I would like to investigate automatic model classification to provide model-specific translations to different reasoners. Such translation can use the full potential of reasoners to deliver the best possible reasoning capabilities. We have already noticed that even slight changes in model translations can have huge impact on analysis time. Efficiency matters for operations such as consistency checking, dead element detection, or model refactoring.

Furthermore, analysis of the full models is often resource-consuming and unnecessary. One way of solving this problem is to discover automatic model slicing methods and then apply model-specific translations. This is a very interesting area because it can make verification of non-trivial code/model properties easier and possible even for large models. The challenge there is to find the smallest slices that preserve semantics and are useful for analysis.

I would also like to invent new analyses for software product lines as they are instrumental for generating correct code and reducing program's size. For example, one would like to ensure that the mapping between the problem and the solution space is complete, i.e. that there is at most one product per product configuration. To show that my work is viable and free from errors, I am planning to evaluate it on big real-world examples of product lines such as variability models of the Linux and eCos kernels, and software product lines used in the automotive industry.

To sum up, I have no doubt that effective reasoning on models requires exploiting their properties. Reasoning is crucial for creating reliable and maintainable software product lines. I look forward to exploring new techniques and implement cutting edge tools that will push the state-of-the-art in software product lines for academia and practitioners alike.