Towards Automated Composition of a Product Line

Shengmei Liu
sliu7@wpi.edu
WPI
Worcester, Massachusetts

George T. Heineman
heineman@wpi.edu
WPI
Worcester, Massachusetts

ABSTRACT

Object-oriented (OO) frameworks represent a significant achievement in extensible design, but there are many well-documented challenges when third-party programmers attempt to use and refactor them. In earlier work, we described how to migrate existing OO framework-based software into a software product line structure using combinatory logic synthesis (CLS) integrated into FeatureIDE, an Eclipse-based IDE that supports feature-oriented software development. While initially successful at synthesizing a few instances of a product line, the approach does not scale to support larger product lines because it does not adequately capture the commonality and inherent variability in the application domain. In this paper, we analyze these problems, and introduce our new approach to construct product line which overcomes many drawbacks of old approach. We describe how application domain modeling helps automated composition, and how our redesigned CLS-engine improves scalability. Results are illustrated by scaling the well-known graph product line with our approach, while reducing the average amount of instance-specific code significantly, generating more readable code and providing more convenience for refactoring.

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Shengmei Liu and George T. Heineman. 2018. Towards Automated Composition of a Product Line. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Software product lines (SPLs) refer to software engineering methods, tools and techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production.

A SPL is a family of related programs. When the units of program construction are features—increments in program functionality or development—every program in an SPL is identified by a unique and legal combination of features, and vice versa. Family member refers to individual product. A variation point represents a decision leading to different variants of an asset. A variation point consists of a set of possible instantiations (legal variations of the variation point). A variation point usually specifies the binding times, that is the time/times at which a decision about the instantiation has to be taken.

The variant derivation is the action in which assets are combined from the set of available assets and contained variation points are bound/instantiated. If there are variation points with multiple binding times, the derivation will happen stepwise at each binding time. The result of such a derivation is a set of derived assets. The derivation can be executed technically in many ways. The simplest way is to copy assets and modify (parts of) them (e.g. the configuration parameters) manually. The result of such a derivation is often called a configuration.

We make the following observations regarding product lines:

–It should be possible to generate an instance of the product line using configuration

–Should be able to add to and remove features from a product line after it has been defined

–A product line definition is still reflected in software artifacts, which means it should support common refactoring functionality

–Should be feature-rich, which means one can potentially envision a significant number of PL members (e.g., not just a handful)

Without one of these characteristics, it could only be considered a closed system.

Here is a reference to Kästner's paper [3].

Here is a reference to [4]. Here is a reference to [7]. Here is a reference to [5]. Here is a reference to [1]. Here is a reference to [6]. Be sure to place where it is discussed [2].

1.1 Approaches to Product Line Development

Software product lines create unique engineering challenges for a number of reasons. First, it should be possible to generate any number of instances of the product line, and by this we mean the code artifacts for the instance application. These are then compiled (or interpreted) to realize the execution of the application. Second, there are multiple development efforts; one can work on code that will effectively be used by all product line instances, but at other times, one is focused on writing code for just a single instance from the PL.

A major goal of featured-oriented PL is to derive a product automatically based on user's selection. There are two approach widely used in practice, annotation-based approach or a composition-based approach, which differ in the way they represent variability in the code base and the way they generate products.

1.1.1 Annotation-based Approach. There is a single body of code artifacts which fully contains all code resources used by all members of the product line. Using different tools or language-specific capabilities, a compiler (or processor) will extract subsets of the code to be used for a PL instance. One of the most common approaches is to

SPLC'19, 9–13 September, 2019, Paris, France

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

use compiler directives embedded within the code as a means for isolating code unique to a subset of product line instances. Then each product line instance can be generated by compiling the same code base with different compiler flags, resulting in different executable instances. Due to the nature of this approach, often one cannot review the source code for individual instances.

In annotation-based approaches, the code of all features is merged in a single code base, and annotation mark which code belongs to which feature. In some sense, an annotation is a function that maps a program element to the feature or feature combination it belongs to.

Annotation-based approaches are widely used in practice because they are easy to use and already natively supported by many programming environments. It keeps good readability and low complexity, however, relatively simple tool support can address scattered code or error.

1.1.2 Composition-based Approach. Another approach is to design a feature tree which is used to capture all the externally visible features that can be used to differentiated one product line instance from another. Then code assets are internally associated with each of these visible features. Finally product line instances are configured by selecting for inclusion features from the feature tree, potentially restricted by constraints. A composition engine processes the code assets associated with the selected features to create the final source code for the product line instance.

Composition-based approaches locate code belonging to a feature or feature combination in a dedicated file, container, or module. A classic example is a framework that can be extended with plug-ins, ideally one plug-in per feature. The key challenge of composition-based approaches is to keep the mapping between features and composition units simple and tractable.

Another way to view the difference between annotation and composition is that annotation separate concerns virtually and composition separate concerns physically,and code is removed on demand with annotation while composition units are added on demand.

1.1.3 Product Line Techniques In Industry. Annotation-based approach are not so widely used as composition-based approach because drawbacks mentioned above. Here are some existing annotation-based approaches:

code coloring (FeatureCIDE): CIDE is an Eclipse plug-in that replaces the Java editor in SPL projects. Developers start with a standard Java legacy application, then they select code fragment and associate them with features from the context menu. The marked code is then permanently highlighted in the editor using a background color associated with the feature

Type checking approach, a product-line-aware type system that statically and efficiently detects type errors in annotation-based product-line implementations.

To work with FeatureIDE, the primary challenge is to design a feature tree model to represent the desired product line application domain. Because features are cross-cutting with regards to the artifacts in the programming language, the various composer engines supported by FeatureIDE accomplish the same goal in a variety of ways.

AHEAD has feature modules for each concrete feature, and the corresponding composition tool places generated source code directly into the Eclipse source folder. AHEAD brings separate tools together and selects different tools for different kinds of files during feature composition,establishing a clear interface to the build system. Composing Jak files will invoke a Jak-composition, whereas composing XML files invokes an XML-composition tool.

FeatureHouse tool suite has been developed that allows programmers to enhance given languages rapidly with support for feature-oriented programming. It is a framework for software composition supported by a corresponding tool chain. It provides facilities for feature composition based on a language-independent model and tool chain for software artifact, and a plug-in mechanism for the integration of new artifact languages.

Deltaj is a Java-like language which allows to organize classes in modules. A program consists of a base module and a set of delta module in a stepwise manner. Much like a feature module, a delta module can add new classes and members as well as extend existing methods by overriding. In contrast to feature modules, delta modules can also delete existing classes and individual members.

In LaunchPad, each feature can contain any number of combinators, designed using a DSL we had developed to simplify the writing of combinators for an earlier CLS tool. A configuration is a subset of features from which a repository is constructed. Each feature can optionally store target definitions, which are aggregated together and then used as the basis for the inhabitation requests, i.e.As each request is satisfied, the synthesized code from the resulting type expression M is stored in the designated source folder.

1.1.4 Evaluation of related work. The key challenge of composition-based approaches is to keep the mapping between features and composition units simple and tractable. Preprocessor-based and parameter-based implementations are often criticized for their potential complexity, lack of modularity, and reduced readability. And they all have some problems which is the hard to refactoring.

In practice most PLs appear "from the ground up" where developers take advantage of language-specific capabilities to annotate different code regions as being enabled (or disabled) based on compiler directives. Starting from an annotation-based code repository many composition-based approaches simply "snipped" or refactored code fragments to recreate countless tiny "features" that could be selected.

Manual composition is a configuration process. A designer selects individual features from a feature model and relies on constraints to ensure the resulting product line member is valid. Manual Composition is limited to a potential total of 2N configurations where N represents the number of available features in the model. There is no domain modeling, What commonly occurs is the designer must make sure that changes to any of the units will not invalidate those product line members that incorporate that feature.

Another problem is that the features are fixed and unchanging. If we need to make some modifications to current instances, we may need to trace all the way back and change the code in many classes because it's inheritance structure. If we want to add features which is slightly different from existing ones, we may need to start from very beginning.

2 A NEW APPROACH IS NEEDED

There are features to represent the structure of a variations, and there is a feature for each variation. Our goal was to support the easy construction of new variations by reusing existing features where possible and adding new features as needed to support the functionality expected of new variations.

2.1 Essential Characteristics

2.1.1 CLS generic composition. With dominated approach for using feature in PL, n features in the feature tree may generate 2n configurations which will become product line instances. But if we use CLS as the algorithm for composition, the fundamental units will be combinators instead of features. The CLS starts with a repository of combinators to which a user issues a query which attempts to find a type in the repository using inferencing.

Combinators can be dynamic and added at composition time, something which is simply not possible in traditional feature trees used by feature-oriented product lines.

To better explain these dynamic combinators, consider having a feature model with a feature that provides variability and there are a number of fixed sub-features that are tailored for each valid variation. For example, "Number of external hard disks" might have sub-features "One-Hard-Disk", "Two-hard-disks" and so on. Individual members of the PL are configured, accordingly, to select the desired number of external hard disks. In contrast, using CLS a single combinator class NumberOfExternalDisks is parameterized with an integer, and one can instantiate a combinator (NumberOfExternalDisks(3)) and add to the repository as needed based on the modeling needs of the member.

Without making 2n configurations, using CLS will significantly simplify code system in PL, optimize code structure make it more readable and reasonable.

2.1.2 Application domain modeling. This appears missing in nearly every approach we see. This happens because Feature Trees do not require any other modeling besides the tree itself, and annotation-based approaches rely solely on the codebase itself.

Because there is no domain modeling, the various Feature-IDE approaches all appear to have configurations which become ineffective domain modeling. For example, in some FeatureIDE models, there is a feature with sub-features that appear to be nothing more than instantiations of different configurations, which makes the width of feature tree in AHEAD huge.

2.1.3 Compositional manipulation. Feature-IDE relies on externally provided composition engines to process code fragments. The challenge is that FeatureIDE can make no semantic guarantees about the resulting code. Also there is no theoretical foundation for the composition, which rather simply is assembly. During assembly, units are wired together without making any changes to the units themselves.

2.1.4 Language agnostic. Without being language limited as normal ways, our approach is more laguage agnostic. Choice of language have been more diversified, which could benefit more engineers with different backgrounds. We don't have to take advantage of single language to build our code base. For example,we have to use .jak files in AHEAD. If you are not familiar with the language, you can't use the approach.

2.1.5 Code sharing between assets. Like we mentioned above, dynamic combinators can be constructed to add methods into classes. Assets can share some basic code, with different methods included.

3 GRAPH PRODUCT LINE

The Graph Product Line (GPL) is a well-known case study within the software product line community. This product line supports variations in a library of graph data strutures and algorithms. A possible feature diagram of the graph library is shown in Figure?. The root is labeled with Gpl to represent a graph product. It has a mandatory child feature GraphType, because each graph library has to implement an type, which is either Directed or Undirected. Furthermore, three other child features of the root are optional: Search, Weighted and Algorithm. Search strategies may be either breadth-first search (BFS) or depth-first search (DFS). Algorithm offers a selection of graph algorithms as child features. Since it's optional, either zero, one, ore more algorithms may be presented in a graph product. In our example, the algorithm for minimal spanning trees MST has two alternative implementations, Prim and Kruskal. Some non-local conditions are modeled as explicit Boolean constraints– for example, minimal spanning tree make only sense for weighted graphs, and shortest paths can be computed directed graphs only.

4 DESIGN OF CLS-BASED GPL

The design of the CLS-based GPL goes here...

5 EVALUATION

Evalution of our approach goes here...

6 CONCLUSION

We conclude in this section...

Also talk about future work

7 ACKNOWLEDGMENTS

Identification of funding sources and other support, and thanks to individuals and groups that assisted in the research and the preparation of the work should be included in an acknowledgment section, which is placed just before the reference section in your document.

This section has a special environment:

```
\begin{acks}
...
\end{acks}
```

so that the information contained therein can be more easily collected during the article metadata extraction phase, and to ensure consistency in the spelling of the section heading.

Authors should not prepare this section as a numbered or unnumbered `\section`; please use the "`acks`" environment.

8 APPENDICES

If your work needs an appendix, add it before the "`\end{document}`" command at the conclusion of your source document.

Start the appendix with the "`appendix`" command:

```
\appendix
```

and note that in the appendix, sections are lettered, not numbered. This document has two appendices, demonstrating the section and subsection identification method.

9 SIGCHI EXTENDED ABSTRACTS

The "`sigchi-a`" template style (available only in LaTeX and not in Word) produces a landscape-orientation formatted article, with a wide left margin. Three environments are available for use with the "`sigchi-a`" template style, and produce formatted output in the margin:

- `sidebar`: Place formatted text in the margin.
- `marginfigure`: Place a figure in the margin.
- `margintable`: Place a table in the margin.

ACKNOWLEDGMENTS

To Robert, for the bagels and explaining CMYK and color spaces.

REFERENCES

[1] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 665–668. https://doi.org/10.1109/ICSE.2015.218

[2] Christian Kästner. 2010. CIDE: Virtual Separation of Concerns (Preprocessor 2.0). http://ckaestne.github.io/CIDE/

[3] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type Checking Annotation-based Product Lines. *ACM Trans. Softw. Eng. Methodol.* 21, 3, Article 14 (July 2012), 39 pages. https://doi.org/10.1145/2211616.2211617

[4] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. 2005. Tool support for feature-oriented software development: featureIDE: an Eclipse-based approach. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. ACM, 55–59.

[5] Florian Proksch and Stefan Krüger. 2014. *Tool support for contracts in FeatureIDE.* Universitäts-und Landesbibliothek Sachsen-Anhalt.

[6] Kirk Sayre. 2005. Usage Model-based Automated Testing of C++ Templates. *SIGSOFT Softw. Eng. Notes* 30, 4 (May 2005), 1–5. https://doi.org/10.1145/1082983.1083277

[7] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-oriented Software Development. *Sci. Comput. Program.* 79 (Jan. 2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

A RESEARCH METHODS

A.1 Part One

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi malesuada, quam in pulvinar varius, metus nunc fermentum urna, id sollicitudin purus odio sit amet enim. Aliquam ullamcorper eu ipsum vel mollis. Curabitur quis dictum nisl. Phasellus vel semper risus, et lacinia dolor. Integer ultricies commodo sem nec semper.

A.2 Part Two

Etiam commodo feugiat nisl pulvinar pellentesque. Etiam auctor sodales ligula, non varius nibh pulvinar semper. Suspendisse nec lectus non ipsum convallis congue hendrerit vitae sapien. Donec at laoreet eros. Vivamus non purus placerat, scelerisque diam eu, cursus ante. Etiam aliquam tortor auctor efficitur mattis.

B ONLINE RESOURCES

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam interdum magna at lectus dignissim, ac dignissim lorem rhoncus. Maecenas eu arcu ac neque placerat aliquam. Nunc pulvinar massa et mattis lacinia.