Augmenting Feature-modeling with Application Domain Modeling

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. 2019. Augmenting Feature-modeling with Application Domain Modeling. In *23rd International Systems and Software Product Line Conference, 9–13 September, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

with application domain modeling for featureIDE and composition engines instead of providing a standard black-box otherwise restrict users' abilities.

Software product lines (SPLs) refer to software engineering methods, tools and techniques for creating a collection of similar software systems, called product line members, from a shared set of software assets using a common means of production. A number of characteristics separate SPLs from routine large, complex systems and thus there is a need for specialized tools and techniques to design, implement and maintain SPLs.

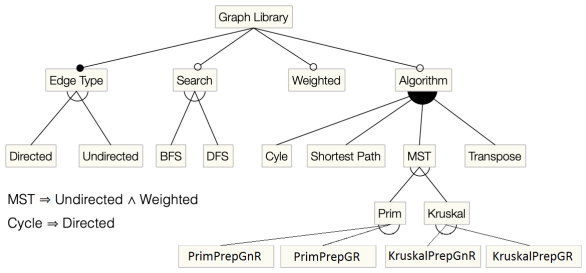

Figure 1: Graph Product Line Feature Model

Fundamental to many SPL approaches is the concept of a *Feature Model*, first introduced by the Feature-Oriented Domain Analysis (FODA) technique [17] and refined over the years by numerous researchers. A feature model is composed of user-visible features from which a wide variety of configurations can be defined through the selection of individual features. Many tool chains have been developed to generate product line members "on demand" from a feature model configuration. Developers need to construct the SPL code artifacts in a specific way that enables the tools to work; this may include, for example, low-level compiler directives embedded in source code, or specialized languages (such as JAK [7] or DeltaJ [27]) that developers use to encode the artifacts.

There remains a potential issue, however, when the variability inherent in the product line relates to the application domain model. The common result is that one must integrate these domain-model concepts into the feature model so they can become part of the configuration process. The feature model from Figure 1 contains features relating to behavior (i.e., **Prim** contains Prim's implementation of the Minimum Spanning Tree problem) and some relate to structure (i.e., **Weighted** captures the notion that edges in a graph may have weights), while a third set (i.e., **KruskalPrepGnR**) are features that mediate between the behavior and structure. As one might expect, since a Feature Model is global in scope (and uniformly decomposed into features), the dominant decomposition can fail to account for the interrelationships of the features.

In past work, we have explored product lines within a number of application domains [8, 15] as we investigate the role that domain modeling should play within an SPLC. In particular, given the extensive developments in model-driven engineering, we looked into ways to integrate a separate domain model into the tool chain used for generating product line instances. We demonstrate how

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

to incorporate domain modeling within the Graph Product Line, a canonical SPLC used extensively within the research community. Our focus is on using code generation to automatically generate the boilerplate code that would otherwise have to be customized based on structural considerations determined by the domain model. Our aim is to make it easier to design the code artifacts for individual features that will be able to compose together regardless of the structure of the domain model. In a way, we are inspired by the underlying philosophy of the Demeter project [22] which aims to separate objects from their operations, to allow each to evolve independently without significant impact on each other.

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 three philosophical approaches widely used in practice – *annotation-based* , *composition-based* or *component-based* – which differ in the way they represent variability in the code base and the tool chains used to generate the product line members.

1.1.1 Annotation-based Approach. An annotation-based SPLC consists of 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 pre-processor) extracts subsets of the code to be used for a PL instance. One of the most common approaches is to 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 [1]. Often the compiler directives manipulate the core data structures or class definitions of the software, for example, adding or removing a class field definition [23].

Annotation-based approaches are widely used in practice because they are easy to use and are already natively supported by many programming environments. While these approaches are easily adopted, the limited (and typically non-extensible) tool support can make it more complicated to maintain an SPLC in the face of constant evolution and added (or removed) features. CIDE [18] 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. Researchers have shown how to type check software product lines to identify and eliminate common errors, such as missing class fields or methods. A product-line-aware type checking system has also been developed to statically and efficiently detects type errors in annotation-based product-line implementations [1, 19].

1.1.2 Composition-based Approach. Another common approach is to design a *feature model tree* to capture the externally visible features that differentiate one product line instance from another. Each feature can include any number of associated code artifacts. A product line instance is 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. The n

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, 30].

1.1.3 Component-based Approach. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. The key idea of a component is to form a modular, reusable unit.

Developing product line by constructing and composing reusable was a common strategy. With domain analysis, developers decided which functionality should be reused across multiple products of the product line designed components accordingly. To derive a product for a given feature selection during application engineering, a developer selects suitable components and then manually writes glue code to connect components for every product individually.

1.1.4 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:

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. [7].

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. [2]

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. [27].

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. [15].

1.1.5 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 difficulty to refactoring. [20].

Although component-based implementations are common in product line practice, they lack the automation potential of feature orientation that we aim at. Deciding when to build a reusable component and what to include in that component is a difficult design decision, one need to have good understanding of whole scope to do that. There is no domain in this approach, glue code is a necessary, it's no more than assembling components. [1]

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

were represented in feature models, we identified a number of limitations in how modeling was represented and devised a number of strategies for improving the way these application domain models could improve the process of generating product line members.

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 Integrated application domain models

The concept of an application domain model is common in software engineering because of its ability to identify real-world concepts that will ultimately be present in a software system. When the model is actively maintained over time, as changes are introduced and the software system evolves, there is the opportunity to take advantage of its high-quality information.

Annotation-based approaches for SPLs that rely solely on the embedded compiler directives within the code are incompatible with a separately designed application domain model. When product lines use compiler directives to alter the very structure of the object-oriented classes in the code base, as observed in [19], there needs to be extensive type checking to ensure only valid product line members are generated. In these cases, one typically finds that "the code is the design" and there no other separate representation of the classes being manipulated by the compiler directives. For this reason, our investigation focused primarily on composition-based SPLC approaches.

Researchers have investigated various object-oriented technologies to support SPLs [6, 14] with the focus of finding ways to develop reusable components within an SPLC, whether using variations of *mixins* [9] or C++ Templates [32]. What is missing, however, is the capability for the application domain model to be fully integrated into the SPLC tool chain and have as great an influence as the features themselves. We observed in nearly every approach using feature models that the lack of a domain model resulted in increasingly complicated feature models. Specifically, numerous sub-features appeared to be nothing more than instantiations of different configurations, as we first identified in Figure 1. This is most evident when reviewing sample product lines as supported by FeatureIDE **??**, an extensible Eclipse-based framework for feature-oriented software development. A product line member is defined solely by the configuration of existing features in the feature model, as allowed by the defined constraints. Because there is no separate domain model, the various FeatureIDE approaches all appear to have configurations which become ineffective domain modeling. For example, in some FeatureIDE models, there are features with sub-features that appear to be nothing more than instantiations of different configurations, which does little to reduce the overall complexity, and instead, simply widens the feature model tree.

2.2 CLS generic composition

With dominated approach for using feature in PLs, 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. Using CLS generic composition, limitations of composition tools are eliminated.

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.

2.2.1 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.

2.2.2 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. [8].

2.2.3 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 [24].Graph is a fundamental topic in computer science and choosing it minimises the requirement of becoming a domain-expert which is a prerequisite in the product line development process. Still it is complex enough to expose the underlying concepts of product lines and their implementation. 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?.Like any other product line, members of the GPL share some common features and differ in certain variant features. 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.

Some of the common and variant features that the GPL members share are identified as follows:

• Common Features: The common features that the GPL applications share are Vertex and Edge. Which means a GPL application must have these features.

• Variant Features: Members across the GPL share some variant features:

– Graph Type: A graph is either directed or undirected. – Weight: Edges in a graph can be weighted with non-negative numbers or unweighted. – Search: A graph application can implement at most one searching algorithm, Breadth First search (BFS), Depth First Search (DFS) or none. – Algorithms: A graph application implements one or more of the following algorithms: Number (Vertex Numbering), Connected (Connected Components), Strongly (Strongly Connected Components), Cycle (Cycle Checking), Shortest (Single-Source Shortest Path) and MST(Minimum Spanning Tree).Details of these algorithms can be found in any algorithm book and more information on GPL can be found at: http://www.cs.utexas.edu/users/dsb/GPL/graph.htm. AND http://stg-tud.github.io/sedc/Lecture/ws16-17/6-SPL.pdf.

Here are papers using GPL [31]. [33]. [26]. [4]. [25]. [16]. [5]. [13]. [21]. [3].

4 DESIGN OF CLS-BASED GPL

Our improved solution successfully integrates application domain modeling with CLS to dramatically improve our ability to automatically compose members of a product line structure. We start by designing a domain model. This domain model simply models aspects of these variations, and define constraints between assets. For example, MST algorithm can only be applied on undirected and weighted graph. Then we construct concrete combinators and dynamic combinators to represent instances, which are added to gamma repository. After targets are identified, the next step is to inhabit. [12]. We instantiate our combinators, perform reflection on them to allow the CLS engine to gather their type information, and then ask CLS to produce all compositions (by invoking apply methods). [15]. Each valid configuration is an instance of the model.

We have achieved the following vision: to introduce a new member of the product line or to modify an existing one –just create (or modify) the application domain model and synthesize the member using automatic composition. When a variation includes unique logic that has not been seen before, then appropriate localized Scala code changes are introduced; if multiple variations could benefit from the variation-specific concepts, then the Scala code is refactored to bring this logic into the shared area.

4.1 More powerful Scala Implementation

By implementing CLS in a general purpose language, we move away from an interpreted DSL towards an embedded DSL that increases expressivity. However, by choosing Scala we gained a number of significant benefits as well:

• Scala program immediately interoperate with numerous Java libraries in the wider software ecosystem. Notably, we integrate JavaParser to manipulate ASTs and manage the synthesized software using jGit to provide API-level access to the Git version control system. • Scala offers expressive string interpolation to embed variable references and entire source code fragments within strings. We use this extensively to make it easy for programmers to write readable combinators. • IntelliJ offers a powerful Scala environment that seamlessly integrates the CLS tool, so programming with combinators is treated the same as programming in a native language. As code is synthesized, it is placed into local Git repositories and then checked out into IntelliJ.

In our approach, we consturct two kinds of combinators with CLS technology. We have extensible static @combinator for all base classes, like Vertex and Graph, which is necessary for the whole product line. Once you add them into repository, they will always be there. Listing below shows few line of vertexBase which synthesize the base class for a vertex:

```
@combinator object vertexBase{
def apply(extensions:Seq[BodyDeclaration[\_]]):
    CompilationUnit = { Java(s"""
       |public class Vertex  {
       |${extensions.mkString("\n")}
       |}""".stripMargin).compilationUnit
}
val semanticType: Type =
  vertexLogic(base, extensions)
  =>: vertexLogic(base, complete)
}
```

The apply method has a parameter which will ultimately be resolved by CLS. It takes a class gives Seq[BodyDeclaration[_]] with vertexLogic(vertexLogic.base,vertexLogic.extensions) as semanticType. A combinator is instantiated from this class–based on a specific application domain object that models the desired variation and added to gamma repository. `${extensions.mkString("\n")}` is where the extra imports, fields, and methods required for the specific variation determined by domain object will be added later by dynamic combinator. The types of the parameters designed by semanticType ensure that the arguments are all appropriate, and so the resulting synthesized class will have no syntax errors.

Instead of relying solely on annotated fixed combinators (as shown in Listing above), we can programmatically add customized dynamic combinators(as shown in Listing below). This ability was a major step forward in our ability to populate with modular units suitable to compose any number of instances from a product line structure. For example ,we want to add color attribute to vertex:

```
class ColoredVertex {
  def apply() : Seq[BodyDeclaration[\_]] = {
    Java(s"""
      |     private int color;
      |
      |     public void setColor(int c) {
      |         this.color = c;
      |     }
      |     public int getColor() {
      |         return this.color;
      |     }
      |""".stripMargin).classBodyDeclarations()
  }
  val semanticType: Type = vertexLogic(base, var_extensi
}
```

By using Scala rather than a DSL we can immediately take advantage of writing reusable and extensible combinators. In AHEAD approach, there are many duplicated code, for example, the refinements to Edge are exactly same in MSTKruskalPrepGr and MSTPRIMPREPGnr. Also some empty methods are there only for future override. If you do not have a solid understanding for the whole scope in the beginning, which happens a lot in industries, it's hard to construct a structure like that. With our approach, it's much easier to construct product lines from scratch, and the code will make more sense. In Ahead approach, some features can only be used for once. For example, only thing feature dProgStronglyConnected do was to provide a run method. This will lead to redundant code, while our approach doesn't have such problems.

Other than adding content to existing class, which can also be realized with DeltaJ, we can also modify content in the class easily by constructing a combinator which takes a CompliationUnit as parameter, customize the class and return it as a CompliationUnit.

With our approach, it makes it possible to customize existing classes by either adding content or just changing it.

4.2 Perform Computations During Synthesis

Once the combinators (both static and programmable) are added to repository, the inhabitation targets are requested and a forest of type expressions is computed that satisfy these requests. To complete the code generation, the apply methods of the respective combinators in the types are computed.

4.3 Map from Application to Solution Domain

Our most successful contribution is the ability to create customizable and extensible code generators that produce solution domain code fragments directly from application domain elements. This capability truly shows how to transform a lightweight application domain model into executable solution domain code, without relying on proprietary or complicated tool support. The resulting product line members are constructed more directly than existing configuration mechanisms.

The automated composition technology reveals the direct links between the application domain and the solution domain while allowing each to be independently evolved and modified. Evolution can occur in the application domain (i.e., constraints can be added or existing ones refactored) or the solution domain (i.e., an API for a framework changes and the corresponding invocations must adapt). Because there is no design tool separating the programmer from seeing and modifying the combinators, our approach provides maximum power and flexibility in designing and integrating variation speific combinators. Note that the code generator registries are an independent contribution of this paper, and their behavior is orthogonal to the CLS algorithm.

4.4 Engineering Product lines

The CLS algorithm is invoked as a locally hosted web service executed from within IntelliJ as a background task. As combinators are designed and constructed, IntelliJ helps programmers with all of the modern conveniences provided by an IDE, such as code completion, syntax hilighting, and searching (both definitions and usage of). Each graph variation is configured using a top-level Scala class, such as shown above. To request construction, visit localhost:9000/simple in a web browser and the designated code executes.

As customized by the application domain model, a repository of static combinators is constructed based on the Scala traits that contain variation-specific combinator definitions.And all programmable combinators are added to the repository and, using reflection, the CLS engine gathers all type information. There are two steps to synthesize a specific graph variation. First, upon activation, the CLS web-service performs type inhabitation on the requests which are derived from the application domain model instance or variation. These targets are programmatically supplied, rather than hard-coded. The result of inhabitation is a forest of type expressions returned to the client invoking the web-service. Second, instead of simply pretty printing the synthesized code , the web-service packages the synthesized code corresponding to the type expressions in a locally-hosted Git repository.

Since all artifacts are programmed natively, we can immediately increase programmer productivity in at least four ways; (1) use breakpoints in IntelliJ within the combinator Scala code to debug step-by-step through the static or programmatic combinators; (2) use standardized logging capabilities (with different warn, debug, and error levels) to generate log files to record the detailed context when debugging complicated combina- tors; (3) incorporate unit tests for individual combinators as well as the POJOs used to model the application domain, so programmers can increase their confidence in the modular units; (4) use code coverage analysis over the test cases to identify non-executing code and potential weak areas;

Using CLS is a major improvement over burying build details within Makefile/Ant scripts which encode the detailed knowledge for constructing individual product line members. Since the entire composition process is transparent, programmers have full control over the process; for example, we have scripts to automatically synthesize and compile all registered solitaire variations. A limitation of FeatureIDE is that the user must manually choose the current configuration, and only one composed product line member can exist at a time. Finally, the CLS composed code is readable, and at first glance is indistinguishable from manually written software.

5 EVALUATION

Evalution of our approach goes here...

6 CONCLUSION

In our approach, all product line family modeling is accomplished in the application domain model space and the combinators are written to simply "materialize" the actual solution-domain code that reflects the application domain model. This work will also enable true independence from underlying technologies and external dependencies, and in particular, address the common problem of upgrading code that depends upon a framework that has evolved.

7 RELATED WORK

variants are the SPL frameworks supported by the most extensive commercial case-studies in our list. A feature they share with CLS is support for domain spaces with non-boolean types (e.g., numbers), which can also be automatically computed. This exceeds the capabilities of feature diagrams, where extensions have been proposed to include numbers for modeling cardinalities .[29]. In our example, variable cardinalities are used to model the different numbers of piles in a game and positions for their layout are automatically computed. Inclusion of non-booleans blows up the order of magnitude of possible products from 2n , where n is the number of decisions to be made, to infinity.

The authors of [2] explains the importance of supporting multiple target languages and one of the main improvements of their FeatureHouse framework over AHEAD [7] is to eliminate language extension overhead. From a BNF grammar with some annotations, FeatureHouse generates a parser and pretty printer for superimposed AST components. Given an ordered sequence of these components, they can be automatically composed to generate an AST, which is then pretty printed to create a target source code artifact. CLS is in some sense complementary to this approach: it automates finding the sequence in which AST manipulations are to be made and leaves their specification to the developer. In principle, the two frameworks could be combined. This seems especially promising for the generation of parsers and pretty printers, which is currently one of the major efforts when adapting CLS to a new target language.

The DOPLER [11] framework avoids any assumptions about target languages and instead operates with a decision-based meta model of code generators. Code generators are automatically executed by the rule-based Drools engine [28] whenever decisions are taken. This offers similar flexibility to combinators in CLS, which can perform arbitrary computation. A downside of the DOPLER framework is that the low level, solution space specific collaboration of code generators has to be coded manually. In CLS, code generating combinators can specify their need for input from other code generators in their type, and the inhabitation algorithm will automatically compose them. In DOPLER this solution space dependency would have to be hard coded or pushed to the domain space to be handled by Drools.

In the broader scope of software product lines beyond a specific framework, a feature-based extension to UML is proposed in [10]. In contrast to our modeling of the domain specific variability with classes, this extension modifies models of solution domain classes depending on Object Constraint Language (OCL) expressions.

8 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.

9 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.

10 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] Sven Apel, Don Batory, Christian Kastner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated.

[2] Sven Apel, Christian Kastner, and Christian Lengauer. 2009. FEATUREHOUSE: Language-independent, Automated Software Composition. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09).* IEEE Computer Society, Washington, DC, USA, 221–231. https://doi.org/10.1109/ICSE.2009.5070523

[3] Randall C. Bachmeyer and Harry S. Delugach. 2007. A Conceptual Graph Approach to Feature Modeling. In *ICCS.*

[4] Ebrahim Bagheri, Tommaso Di Noia, Azzurra Ragone, and Dragan Gasevic. 2010. Configuring Software Product Line Feature Models Based on Stakeholders' Soft and Hard Requirements. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10).* Springer-Verlag, Berlin, Heidelberg, 16–31. http://dl.acm.org/citation.cfm?id=1885639.1885642

[5] Ebrahim Bagheri, Tommaso Di Noia, Dragan Gasevic, and Azzurra Ragone. 2012. Formalizing interactive staged feature model configuration. *Journal of Software: Evolution and Process* 24, 4 (2012), 375–400. https://doi.org/10.1002/smr.534 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.534

[6] Don Batory, Rich Cardone, and Yannis Smaragdakis. 2000. *Object-Oriented Frameworks and Product Lines.* Springer US, Boston, MA, 227–247. https://doi.org/10.1007/978-1-4615-4339-8_13

[7] Don S. Batory. 2004. Feature-oriented programming and the AHEAD tool suite. *Proceedings. 26th International Conference on Software Engineering* (2004), 702–703.

[8] Jan Bessai, Boris Düdder, George Heineman, and Jakob Rehof. 2018. Towards Language-independent Code Synthesis, Poster/Demo. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM).* https://popl18.sigplan.org/event/pepm-2018-towards-language-independent-code-synthesis-poster-demo-talk-

[9] Gilad Bracha and William Cook. 1990. Mixin-based Inheritance. *SIGPLAN Not.* 25, 10 (Sept. 1990), 303–311. https://doi.org/10.1145/97946.97982

[10] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06).* ACM, New York, NY, USA, 211–220. https://doi.org/10.1145/1173706.1173738

[11] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. 2011. The DOPLER Meta-tool for Decision-oriented Variability Modeling: A Multiple Case Study. *Automated Software Engg.* 18, 1 (March 2011), 77–114. https://doi.org/10.1007/s10515-010-0076-6

[12] Boris Düdder, Oliver Garbe, Moritz Martens, Jakob Rehof, and Pawel Urzyczyn. 2013. Using Inhabitation in Bounded Combinatory Logic with Intersection Types for Composition Synthesis. In *ITRS.*

[13] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. 2012. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *Advanced Information Systems Engineering*, Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanislaw Wrycza (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 613–628.

[14] Martin Griss (Ed.). 1999. *Implementing Product-Line Features with Component Reuse.* Lecture Notes in Computer Science, Vol. 1844. Springer. https://doi.org/10.1007/978-3-540-44995-9_9

[15] George Heineman, Armend Hoxha, Boris Düdder, and Jakob Rehof. 2015. Towards Migrating Object-oriented Frameworks to Enable Synthesis of Product Line Members. In *Proceedings of the 19th International Conference on Software Product Line (SPLC '15).* ACM, New York, NY, USA, 56–60. https://doi.org/10.1145/2791060.2791076

[16] Martin Fagereng Johansen, Franck Fleurey, Mathieu Acher, Philippe Collet, and Philippe Lahire. 2010. Exploring the Synergies Between Feature Models and Ontologies. In *SPLC Workshops.*

[17] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report. Carnegie-Mellon University Software Engineering Institute.

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

[19] 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

[20] Jongwook Kim, Don Batory, and Danny Dig. 2017. Refactoring Java Software Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A (SPLC '17).* ACM, New York, NY, USA, 59–68. https://doi.org/10.1145/3106195.3106201

[21] Sebastian Krieter and Gunter Saake. 2016. An Efficient Algorithm for Feature-Model Slicing.

[22] Karl J. Lieberherr. 1996. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston. ISBN 0-534-94602-X.

[23] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10).* ACM, New York, NY, USA, 105–114. https://doi.org/10.1145/1806799.1806819

[24] Roberto E. Lopez-Herrejon and Don S. Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering (GCSE '01).* Springer-Verlag, London, UK, UK, 10–24. http://dl.acm.org/citation.cfm?id=645418.652082

[25] Roberto E. Lopez-Herrejon and Alexander Egyed. 2012. Towards Fixing Inconsistencies in Models with Variability. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '12).* ACM, New York, NY, USA, 93–100. https://doi.org/10.1145/2110147.2110158

[26] Roberto Erick Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Evelyn Nicole Haslinger, Alexander Egyed, and Enrique Alba. 2014. Towards a Benchmark and a Comparison Framework for Combinatorial Interaction Testing of Software Product Lines. *CoRR* abs/1401.5367 (2014).

[27] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented Programming of Software Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10).* Springer-Verlag, Berlin, Heidelberg, 77–91. http://dl.acm.org/citation.cfm?id=1885639.1885647

[28] Andy Schürr, Manfred Nagl, and Albert Zündorf (Eds.). 2008. *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers.* Lecture Notes in Computer Science, Vol. 5088. Springer. https://doi.org/10.1007/978-3-540-89020-1

[29] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. 2016. Extending Feature Models with Relative Cardinalities. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16).* ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/2934466.2934475

[30] 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

[31] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *Proceedings of the 2011 15th International Software Product Line Conference (SPLC '11)*. IEEE Computer Society, Washington, DC, USA, 191–200. https://doi.org/10.1109/SPLC.2011.53

[32] Michael VanHilst and David Notkin. 1996. Using C++ Templates to Implement Role-Based Designs. In *Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software (ISOTAS '96)*. Springer-Verlag, London, UK, UK, 22–37. http://dl.acm.org/citation.cfm?id=646898.710025

[33] Zhang., Liya Chakma, and Hongyu Zhang. [n. d.]. Graph Product Line EVALUATING PRODUCT LINE TECHNOLOGIES : A GRAPH PRODUCT LINE CASE STUDY.

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.