The Thesis Committee for Jongwook Kim
Certifies that this is the approved version of the following thesis:

Paan: A Tool for Back-Propagating Changes to Projected Documents

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Don S. Batory

Dewayne E. Perry

Paan: A Tool for Back-Propagating Changes to Projected Documents

by

Jongwook Kim, B.E.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

The University of Texas at Austin

May 2011

Dedication

This thesis is dedicated to my trusted friend, Jungsub Lee.

Acknowledgements

It is my privilege to have Professor Don Batory as my advisor for my graduate work. Without his generous support, guidance, and patience, this work would not have been possible.

I would also like to thank Professor Dewayne Perry for his valuable instructions and comments.

Last but not least, I thank my parents. This thesis is a product of their unconditional love, support, and encouragement throughout the years. Words alone cannot convey my appreciation and love for them.

May 2011

Abstract

Paan: A Tool for Back-Propagating Changes to Projected Documents

Jongwook Kim, M.S.C.S.

The University of Texas at Austin, 2011

Supervisor: Don Batory

Research in *Software Product Line Engineering (SPLE)* traditionally focuses on product derivation. Prior work has explored the automated derivation of products by module composition. However, it has so far neglected propagating changes (edits) in a product back to the product line definition. A domain-specific product should be possible to update its features locally, and later these changes should be propagated back to the product line definition automatically. Otherwise, the entire product line has to be revised manually in order to make the changes permanent. Although this is the current state, it is a very error-prone process. To address these issues, we present a tool called *Paan* to create product lines of MS Word documents with back-propagation support. It is a diff-based tool that ignores unchanged fragments and reveals fragments that are changed, added or deleted. Paan takes a document with *variation points (VPs)* as input, and shreds

it into building blocks called *tiles*. Only those tiles that are new or have changed must be updated in the tile repository. In this way, changes in composed documents can be back-propagated to their original feature module definitions. A document is synthesized by retrieving the appropriate tiles and composing them.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Feature Oriented Software Development (FOSD) is the study of feature modularization and composition for program synthesis in *Software Product Lines (SPLs)*, where a *feature* is an increment in program development or functionality [2]. In FOSD, a *feature module* encapsulates changes that are made to a program in order to add a feature's capability or functionality. Starting with an empty program, adding (or composing on) such modules synthesizes a distinct program in an SPL. Each program in an SPL has a unique feature composition [4, 11, 34].

A hallmark of FOSD is that it takes a compositional approach to program synthesis. Recent progress in FOSD tooling has taken a projectional approach, which is particularly well-suited for decomposing legacy applications into feature modules. The idea is to color or paint a program. All code that belongs to the `Yellow` feature is painted yellow; all code that belongs to the `Red` feature is painted red. If the `Yellow` feature is required, and `Red` is not, `Red` is projected from the program. Painting is a reincarnation of "sysgen" – the use of preprocessors to eliminate unneeded code. The difference is that painting works with abstract syntax tree representations, so that the legality of the projected program's structure can be guaranteed. Stated differently, painting is a disciplined use of `#def` and `#ifdef-#endif` concepts.

Painting also goes further in that it gives a visual way to understand feature interactions [31]. When yellow code is nested inside red code, we see the interaction of the `Yellow` and `Red` features. In this case, how `Yellow` changes the code of `Red`. Of course, there is symmetry: red code that is nested inside yellow shows how the `Red` feature changes the code of `Yellow`. These are examples of two-way interactions. Color nesting *n*-levels deep represents an *n*-way feature interaction. The benefit here is that the relationship between features and feature interactions is not well-understood, and painting provides an attractive way to improve this situation.

Against this backdrop, this thesis takes painting in several new directions. First, we apply the ideas to MS Word documents. We have created a tool, called *Paan* (which is Korean for 'version'), that allows Word documents to be painted. We leverage existing MS Word annotation capabilities to designate (nested) regions of color. Second, painting is presently understood as nested `#ifdef-#endif` regions. Paan not only supports this painting, but also another that arises in programming – *wrapping* (as in method wrapping). Oddly, wrapping is difficult to express using `#ifdef-#endif` because the changes to the region of code that is to be made is outside, rather than inside, that region. (Wrapping envelopes a code region, rather than modifying its internals). Paan supports wrapping natively. Third, Paan implements an algebra (called a *Tile Algebra*) that represents a formal model of painting. And it was through this algebra that the following scenario was envisioned. Suppose a Word document is painted. This document has sensitive data, so only projections (which are themselves Word documents) can be given to others. Now, recipients will want to make changes to their copy. A facility is needed to automatically back-propagate changes made in projected documents to the original painted document. This ability is the key novelty of Paan.

In the following sections, we give a more detailed overview of Paan, this area of research, and the problems to be addressed in this thesis. We start by illustrating the concept of painting.

1.1 PAINTING PROGRAMS

Consider a counted stack [27], where characters are pushed and popped from a String and the number of elements on the stack is counted. Such a stack has three features: `Base`, `Stack`, and `Counter` (Figure 1.1). The `Base` feature is painted with a clear color and represents an empty stack class. The `Stack` feature is painted green and contains the standard push, pop, empty, and top methods, along with a String that encodes the character stack. The `Counter` feature is blue and contains an integer counter and size method. `Stack` and `Counter` interactions are blue inside green, which reset, increment, and decrement the counter variable.

```
class stack {
   int ctr = 0;
   int size() {
      return ctr;
   }

   String s = new String();
   void empty() {
      ctr = 0;
      s = "";
   }
   void push(char a)  {
      ctr++;
      s = String.valueOf(a)
              .concat(s);
   }
   void pop() {
      ctr--;
      s = s.substring(1);
   }
   char top() {
      return s.charAt(0);
   }
}
```

Figure 1.1: The Counted Stack

1.2 TILES AND PROJECTION

The structure of a painted document can be understood in terms of tiles. The BASE tile represents the code of the `Base` feature. The blue-inside-clear (and clear-inside-blue, which in this example doesn't exist) region represents the interaction of `Base` and `Stack` features. This region is labeled BASE#STACK (the order of features in a #-expression does not matter: BASE#STACK = STACK#BASE). There are other tiles in our example, namely BASE#COUNT and BASE#STACK#COUNT (blue-inside-green-inside-clear above). The entire document (Doc) is produced by composing these tiles:

Doc = BASE#STACK#COUNT·BASE#COUNT·BASE#STACK·BASE

A projection of this document that eliminates, say the `Counter` feature, removes tiles whose name includes COUNT. So a projection of Doc without the `Counter` feature yields DocNoC:

DocNoC = BASE#STACK·BASE

1.3 TILE IMPLEMENTATIONS AND VARIATION POINTS

Internally, here is how a program (or Word document) is structured. A program can have any number of labeled VPs, i.e. points at which a code fragment can be inserted. A tile can contain fragments that are to be inserted at VPs. Figure 1.2 shows the counted stack and its five VPs indicated by stars. Each VP is associated with precisely one fragment, which is installed or uninstalled depending on the tiles that are composed. The `Base` feature has a single tile with two variation points VP1 and VP2. The blue tile inside clear contains the fragment that is installed at VP1. The green tile inside clear contains the fragment that is installed at VP2. This fragment has three variation points

VP3, VP4, and VP5. The blue tile inside green contains the three fragments that are installed at these points.

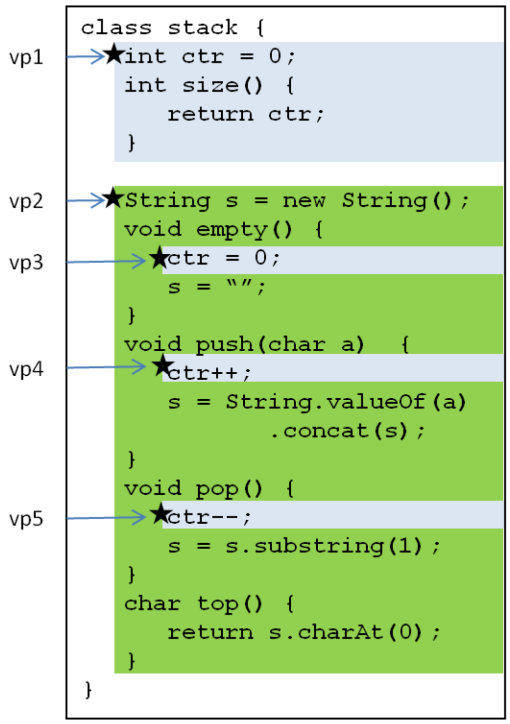


Figure 1.2: The Counted Stack with Variation Points

VPs and fragments are always in one-to-one correspondence [6]. It is not possible for multiple fragments to be installed at the same VP. However, it is possible for some fragments of a tile to remain uninstalled after composition, as they are installed later when another tile adds the required VPs.

In the following chapters, we explore in greater detail these ideas.

1.4 BACK-PROPAGATION OF CHANGES

We assume that a programmer would see markers in a program for existing VPs, and would add new markers and their fragments to add new VPs. Further, a programmer would be at liberty to change any fragment present in the program. If the program source in a SPL is revised to fix bugs locally, updating of the product line based on the local changes should be automatic. Otherwise, the entire product line has to be corrected manually in order to make the fix permanent. As mentioned earlier, is may be undesirable to allow the access to the entire SPL that can contain proprietary data exposed only to certain communities.

In [9], a Tile Algebra suggested a solution. A programmer requests program $P = T_1 \cdot \ldots \cdot T_n$, where P is a composition of tiles. The programmer manually modifies P to produce program $Q = T_0 \cdot T'_1 \cdot \ldots \cdot T_n$. When the client submits the updated program Q, a tool can solve for the changes ΔP. The way this is accomplished is to use a special property of tiles called *involution*: a tile is the inverse of itself ($T \cdot T = 1$). Thus:

$$\begin{aligned} \Delta P \cdot P &= Q && \text{// given} \\ \Delta P \cdot P \cdot P &= Q \cdot P && \text{// compose P to both sides} \\ \Delta P &= Q \cdot P && \text{// involution} \\ \Delta P &= T_0 \cdot T'_1 \cdot \ldots \cdot T_n \cdot T_1 \cdot \ldots \cdot T_n && \text{// substitution} \\ \Delta P &= T_0 \cdot T'_1 \cdot T_1 && \text{// involution} \end{aligned}$$

This is essentially program (or document) differencing. Paan takes a MS Word document with VPs as input and shreds it into tiles. Only those tiles that are new or have changed are updated in the tile repository. Therefore, changes in composed documents can be back-propagated to their original feature module definitions.

Chapter 2: Design

2.1 Office Open XML

Office Open XML is an open standard of XML schemas adopted by Microsoft Office for its default file format [17, 38]. It specifies a compressed, XML-based encoding of Microsoft Office 2007 and 2010 documents, where different XML formats are used for Word, Visio, Excel, and InfoPath. In transition from binary file formats to XML-based representations, MS Office documents are universally accessible across disparate systems by supporting openly available technologies – XML and ZIP compression. The XML schema for MS Word is standardized in ECMA-376 and ISO/IEC 29500, and is available under a royalty-free license [41]. Also, ZIP archives use an industry-standard compression format to allow non-Microsoft products to extract and manipulate MS Office documents [38]. By changing a `.docx` file to `.zip`, the contents of an MS Word document (consisting of multiple XML files and directories) become visible. Above all, Office Open XML is suited for projectional approaches in SPLs, which necessitate mechanisms to explicitly define VPs in a document. We use MS Word's Custom Markup facility to allow users to color Word documents.

2.2 Tagging Features

We created a tool, called *Paan* – Korean for 'version', that enables us to explore a new implementation of coloring, based on the Tile Algebra as the foundation for its design. Specifically Paan works with MS Word documents and relies on the Custom XML Markup facility of MS Word to define nested regions of color and VPs. A markup tag is used to assign a feature name to a fragment of a Word document. A fragment is identified by enclosing start and end tags. In Figure 2.1a, a pair of tags named `blue` surrounds a “Hello World” fragment. Its XML representation is given in Figure 2.1b.

Figure 2.1: MS Word Custom Markup Tags and its XML

2.3 NESTED PREPROCESSOR SEMANTICS

In Paan, tags are nested like preprocessor `#ifdef-#endif` declarations. Projection works in an obvious manner. An inner tag can appear only if all of its enclosing tags (features) have been selected. In Figure 2.2a, `red` tags wrap vowels. Being surrounded by a `blue` tag, they can appear only when both the `blue` and `red` features are selected. Figure 2.2b is a projection where the `blue` feature was not selected. A VP is marked by a special tag named **_reserved_**, and assigned a unique number for identification. A VP's ID number is stored as the tag's property. Figure 2.2c shows another projection where `blue`, but not `red`, was selected.

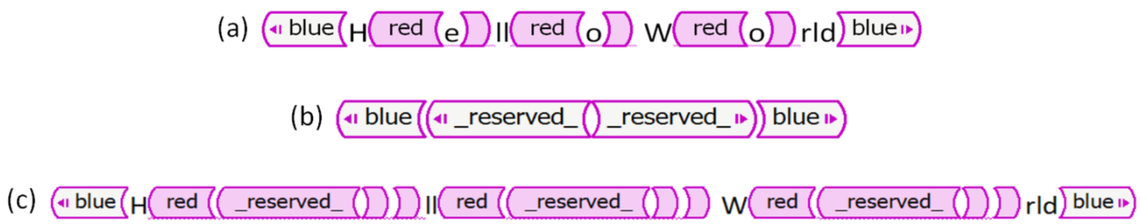


Figure 2.2: Nesting and Projection of Tags

Admittedly, Word's Custom Markup Facility does not have the prettiest or the most compact esthetics. We discuss later our experiences in using this facility.

2.4 Wrapping and Wrappers

Paan extends the coloring technique described above to also support wrapping. A *wrapper* is a fragment that surrounds another fragment. Wrappers occur in AHEAD and FeatureHouse as the way features extend methods [4, 7], in object-orientation where subclasses extend methods of a superclass by wrapping, and in AOP as around advise of execution pointcuts of individual methods [26]. Figure 2.3a shows a base method `m()`. Figure 2.3b shows a refinement of `m()` in AHEAD syntax that wraps `m()`. Figure 2.3c shows the identical refinement of `m()` in AspectJ syntax. Figure 2.3d is the result of composing the base method with this refinement.

```
(a) void m() { counter++; }
(b) void m() { print("before" + counter);
               Super.m();
               print("after" + counter); }
(c) void around() : execution( void C.m() ) {
               print("before" + counter);
               proceed();
               print("after" + counter); }
(d) void m() { print("before" + counter);
               counter++;
               print("after" + counter); }
```

Figure 2.3: Variants of Wrappers

Wrapping is hard to express in preprocessor semantics as it has exactly the opposite semantics of nesting. Let `B` be a base fragment and `W` be a wrapper of `B`. If `B` and `W` are also the names of their respective features, `B` belongs to the `B` tile and wrapper `W` belongs to the interaction tile `W#B`. Unlike nesting, where an interaction tile `T#B` that modifies `B` is fully enclosed by `B`, wrapping reverses the roles where the wrapped tile `B` is fully inside the interaction tile `W#B`. Figure 2.4a shows how a base-wrapper (`BASE`) and refinement-wrapper (`RED`) are colored in Paan. Wrapper tags, `BASE` and `RED`, are in upper-case whereas non-wrapper tags are in lower-case. Figure 2.4b is a projection where `BASE`, but not `RED`, was selected. (Note that `BASE` belongs to the `BASE` tile; `RED` belongs to the interaction tile `RED#BASE)`. Figure 2.4c is a projection where the `BASE` feature was not selected. (The same result would be produced whether or not the `RED` feature was selected, as both `BASE` and `BASE#RED` are projected).

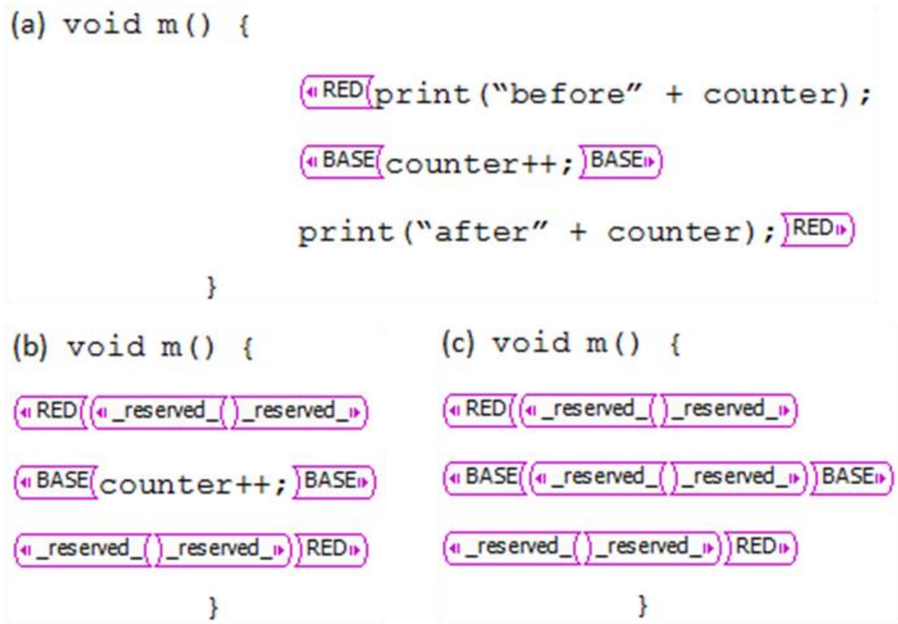
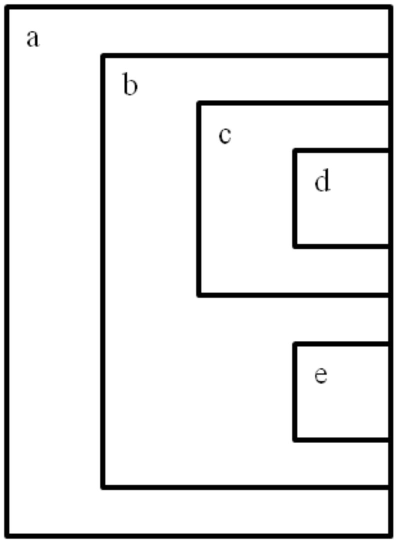


Figure 2.4: Wrappers

2.5 N-WAY INTERACTIONS

Paan offers a visually simple way to recognize *n*-way interactions by the nesting of *n* tags. So an interaction module `f#g#h` would be the set of all fragments that are nested 3-deep using any permutation of features `f`, `g`, and `h`. In practice, 2-way interactions are common, but 3-way interactions arise occasionally. 4-way or higher-order interactions seem rare. Figure 2.5a is a 4-deep interaction of non-wrappers, and Figure 2.5b is list of their user-assigned predicates. In predicate expression of feature interactions, # is mapped to ∧-operation in *conjunctive normal form (CNF)*.

(a)

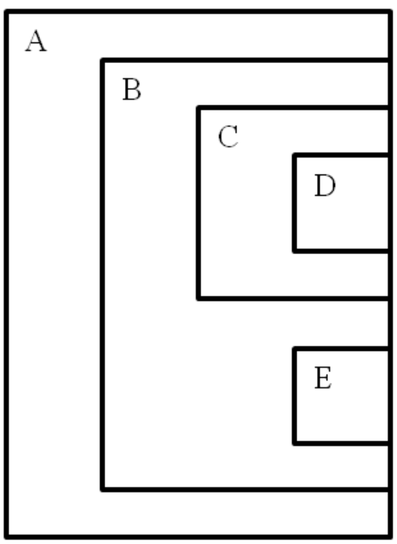


(b)

Tile	Predicate
a	a
b	a∧b
c	a∧b∧c
d	a∧b∧c∧d
e	a∧b∧e

Figure 2.5: Non-Wrapper Interaction and Predicate

(a)



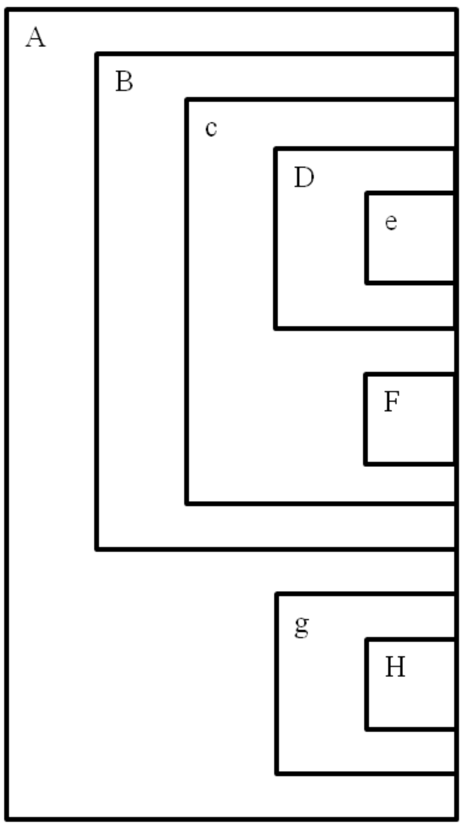
(b)

Tile	Predicate	Base-Wrapper
A	A∧(D∨E)	No
B	B∧(D∨E)	No
C	C∧D	No
D	D	Yes
E	E	Yes

Figure 2.6: Wrapper Interaction and Predicate

Paan also enables higher-order wrappers by allowing users to define a predicate and hence the tile-interaction expression of a wrapper, so that all interaction tiles permitted by the Tile Algebra can be expressed. Figure 2.6a is an interaction of wrappers. Base-wrappers are always the inner-most fragments. Others interact only with the base-wrapper, and need, at least, one base-wrapper to appear. Accordingly, in Figure 2.6b, the predicates are totally different from those of non-wrappers in Figure 2.5b.

(a)



(b)

Tile	Predicate	Base-Wrapper
A	$A \wedge (D \vee F \vee H)$	No
B	$B \wedge (D \vee F)$	No
c	$B \wedge (D \vee F) \wedge c$	-
D	D	Yes
e	$D \wedge e$	-
F	F	Yes
g	$A \wedge g$	-
H	H	Yes

Figure 2.7: Tile Interaction and Predicate

Paan allows a mixture of non-wrapped and wrapped regions. Figure 2.7a illustrates a mixture of regions, and Figure 2.7b lists their predicates. Here is the rule that we use (and that we determined makes the most sense) regarding how to interpret an interaction of region: Wrappers take precedent. Once a wrapper region is determined, this region can be further subdivided by non-wrappers. Non-wrappers extend only to the boarder of its wrapper, *but no further*. In Figure 2.7a, non-wrapper, `c` subdivides wrapper `B`. Non-wrapper `e` subdivides wrapper `D`. And non-wrapper `g` subdivides wrapper `A`.

2.6 PROJECTION

2.6.1 File Projection

Paan represents all subdirectories and files in an internal repository, and allows users to create projections by selecting desired features [21]. Starting from the root directory in a repository, predicates of subdirectories are evaluated by traversing the directory tree in in-order (parent-to-children). Projection empties directories that have a false predicate, and sets them to be invisible and empty. For the files whose predicate is false, projection changes them to be and empty hidden file. Only for MS Word documents with a true predicate is projection on inner tags is performed and (a typically) non-empty file is produced.

2.6.2 Tag Projection

Projection on nested tags works like `#ifdef-#endif` in preprocessors. An inner tag can appear only if all of its enclosing tags have been selected. Projection of wrappers is accomplished in a slightly different way. Paan traverses the Word document `W` in its repository in its entirety. Let `p` denote the set of features that were selected, meaning that their fragments are to remain after projection. A traversal of `W` encounters a sequence of fragments. Let `T` be a fragment and `T(x)` be its propositional formula. If `T(p)` is true, `T` is present in W_p. Otherwise, `T` is not included, but the traversal of `T` to the next fragment continues. This is different than a document with only nested colors, as once a fragment is eliminated, there is no need to search inside the fragment. The need to continue searching further is required as outer wrappers may not appear, but inner wrappers may appear in a projection. Not surprisingly, wrappers increase slightly the complexity of the projection algorithm.

To illustrate, Figure 2.8a shows a `base` fragment wrapped by a `blue` and `green` fragment. Figure 2.8b shows the projection of the `base` feature. Figure 2.8c shows the projection of the `base` and `green` features, and Figure 2.8d the projection of `base` and `blue` features.

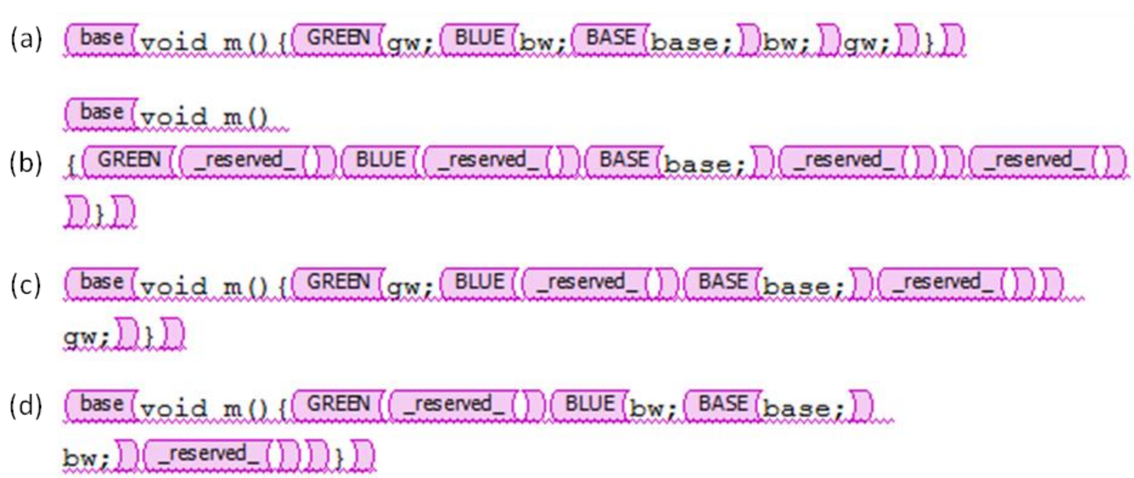


Figure 2.8: Projecting Wrappers

2.7 BACK-PROPAGATION

The key novelty of Paan is that it allows users to edit a projected document, and then merge its changes with the version in the repository. Back-propagation restores the contents of projected VPs by restoring directories, files, and Word fragments. The projected directory or file is simply replaced with the original. Inside a document, VPs indicated by the **_reserve_** tag have their projected contents restored. However, once a VP is deleted by users, installation for that VP is not possible. Moreover, in case that a single wrapper has multiple VPs, one lost VP invalidates all others. In Figure 2.9a, a RED fragment is wrapped by a BLUE fragment. Figure 2.9b shows a projection where the RED feature was not selected. The only condition to restore two VPs of the BLUE wrapper is existence of both.

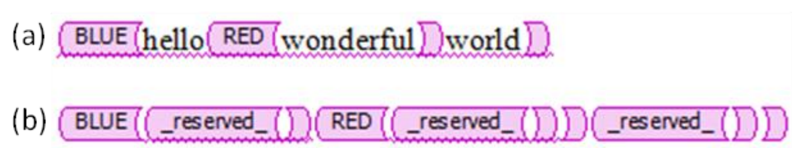


Figure 2.9: Multiple Variation Points of Single Wrapper

Let W be a tagged MS Word document and let W_p be a projection of W, where p is a set of features. Therefore, W_p eliminates all fragments from W whose set of colors do not belong to p. A user can now modify W_p at will, adding new VPs that are instantiated with their text, modifying visible fragments, and deleting existing VPs including VPs whose text has been projected.

To back-propagate the changes in W_p to W, Paan maintains a copy of W in its repository that existed prior to projection. It then traverses W_p to locate VPs whose fragments have been projected. For each such VP `i`, it finds fragment `i` in `W` and restores that fragment in W_p. At the end, all projected fragments in W_p have been restored with their original contents. Paan then discards the original copy `W` and replaces it with W_p. And the projection-back-propagate cycle continues. Here, the restoration of projected VPs can be accomplished in linear time, since a single pass through `W` is enough to find all (VP, fragment) pairs and a single pass through W_p can restore projected VPs.

Paan's back-propagation algorithm is slightly different than that given in Section 1.4. Paan simply assumes that all fragments in W_p have been modied, and proceeds to update its repository copy on this conservative basis simply because it is faster. However, it does use the diffing idea of Section 1.4. A Paan repository can consist of multiple Word documents and directories. If a Word document has not been changed, Paan does not update the repository's copy. Paan infers this information by examining a Word document's revision number and comparing it to the revision number in the repository. If they are the same, the document has not been modified.

2.8 MERGING TILES

Paan supports #-involution (`R#R=1`) in the Tile Algebra. When Paan sees replicated features in region names, it merges regions. For example, when Paan recognizes a region whose #-expression is `R#B#R` (red-inside-blue-inside-red) as in Figure 2.10a, Paan merges `R#B#R` into `B#R` in Figure 2.10b.

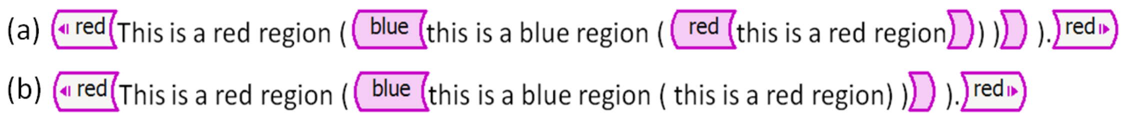


Figure 2.10: Tile Merging

During projection, redundant tags are removed since their predicates are always true. Tile merging is applied to wrappers as well in the following rules:

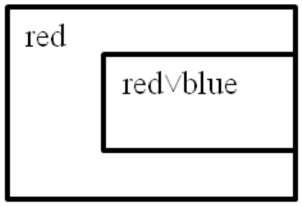
		Inner	
		Non-Wrapper	Wrapper
Outer	Non-Wrapper	Implicative	(Unavailable)
	Wrapper		Equivalent

Table 2.1: Tile Mergence Rules

An inner non-wrapper is merged with its adjacently outer fragment (any of wrapper or non-wrapper) if the predicate of the outer fragment implies that of the inner one. This follows in that an inner non-wrapper can display only if the outer appears, since their predicates are equivalent by #-involution as shown in Figure 2.11.

In case of two adjacent wrappers, they should have equivalent predicates to be merged. That is because wrappers are independent each other but share all or some base-wrappers. Once two adjacent wrappers have the same combination of base-wrappers, they are identical in terms of predicates. In Figure 2.12a, two `RED` wrappers cannot be merged due to different base wrappers. Unlike non-wrappers, the predicates of adjacent wrappers are possibly different although their features are identical as `RED` in Figure 2.12b.

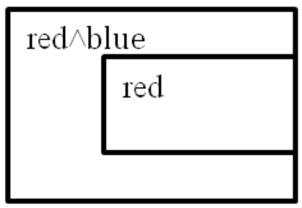
(a)



(b)

Tile	Predicate
red	red
red∨blue	red∧(red∨blue)=red

(c)

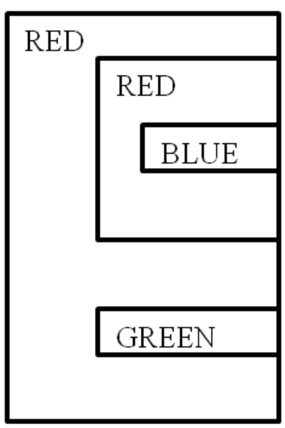


(d)

Tile	Predicate
red∧blue	red∧blue
red	red∧(red∧blue)=red∧blue

Figure 2.11: Tile Mergence for Inner Non-Wrappers

(a)



(b)

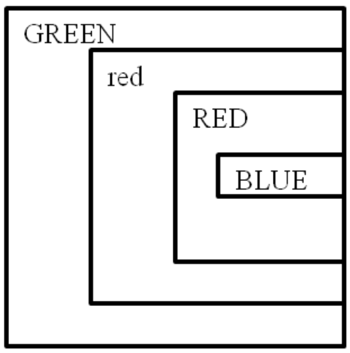
Tile	Predicate	Base-Wrapper
RED	RED∧(BLUE∧GREEN)	No
RED	RED∧BLUE	No
BLUE	BLUE	Yes
GREEN	GREEN	Yes

Figure 2.12: Tile Mergence of Wrappers

Exceptionally, it is not allowed to merge an outer non-wrapper with its adjacent inner wrapper. In Figure 2.13a, the outer non-wrapper `red` implies the inner wrapper

RED but merging both tiles is not possible. The red belongs to its wrapper GREEN, and has no interaction with the wrapper RED as predicates listed in Figure 2.13b.

(a)



(b)

Tile	Predicate	Base-Wrapper
GREEN	GREEN^BLUE	No
red	GREEN^BLUE^red	-
RED	RED^BLUE	No
BLUE	BLUE	Yes

Figure 2.13: Invalid Tile Mergence

Chapter 3: Evaluation

3.1 EXPERIENCE

An Office Open XML document is composed of a series of *parts* and relationships between the `parts` that are stored in a container called a *package*. For instance, a document of pictures roughly consists of two `parts`: one `part` of an XML markup to represent the document and another `part` to provide the pictures.

A MS Word's *Main Document* `part` is encapsulated by a *body* `element` that contains a collection of block-level structures: *paragraph*, *run*, and *text*. The `body` consists of a sequence of `paragraphs`. Also, a `paragraph` contains one or more `runs`, where a `run` is a container for one or more pieces of `text`. Therefore, there exist hierarchical constraints in that `text` must be contained within one or more `runs`, and a `run` must be contained within a `paragraph`. Unfortunately, these syntactic structures can be broken by back-propagation.

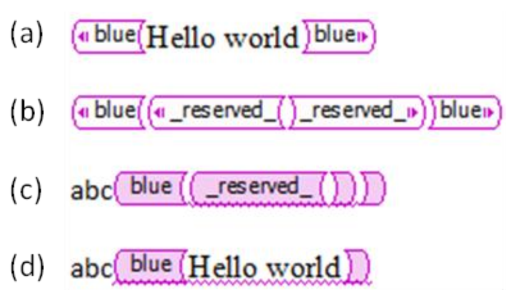


Figure 3.1: Back-Propagation Error

In Figure 3.1a, a pair of tags named `blue` surrounds a "Hello world" fragment. Figure 3.1b is a projection where the `blue` feature was not selected. An arbitrary string "abc" is appended before the VP in Figure 3.1c. We expect that back-propagation restores the “Hello world” fragment at the VP and produces Figure 3.1d. Unfortunately, the resulting Word file invalidates schema conformity.

The XML representation of Figure 3.1a is given in Figure 3.2a. A `paragraph` contains a `run`, and the `run` surrounds a `text` “Hello world”. In Figure 3.2b, this `paragraph` is replaced with the corresponding VP by projection. In Figure 3.2c, concatenating “abc” makes a new `paragraph` to surround the string followed by the VP. Then, back-propagation results in reiterated `paragraphs`: a `paragraph` is in another `paragraph`. This violates the `element` structure, and MS Word creates an error message about syntactically incorrect XML codes.

```
(a) <w:customXml w:uri="sampleSchema" w:element="blue">
      <w:p>
        <w:r>
          <w:t>Hello world</w:t>
        </w:r>
      </w:p>
    </w:customXml>
(b) <w:customXml w:uri="sampleSchema" w:element="_reserved_">
      <w:customXmlPr>
        <w:attr w:name="type" w:val="1234">
      </w:customXmlPr>
    </w:customXml>
(c) <w:p>
      <w:r>
        <w:t>abc</w:t>
      </w:r>
      <w:customXml w:uri="sampleSchema" w:element="blue">
        <w:customXml w:uri="sampleSchema" w:element="_reserved_">
          <w:customXmlPr>
            <w:attr w:name="type" w:val="1234">
          </w:customXmlPr>
        </w:customXml>
      </w:customXml>
    </w:p>
```

Figure 3.2: Block-Level Structures in XML Codes

```
<w:p>
  <w:r>
    <w:t>abc</w:t>
  </w:r>
  <w:customXml w:uri="sampleSchema" w:element="blue">
    <w:r>
      <w:t>Hello world</w:t>
    </w:r>
  </w:customXml>
</w:p>
```

Figure 3.3: Block-Level Error Correction

In Figure 3.3, Paan recovers those errors which break the block-level structures among `body`, `paragraph`, `run`, and `text`, so that the behavior is exactly what users would expect (as in Figure 3.1d). However, a `body element` can contain other block-level contents such as `tables`, `section properties`, `comments`, `revision markers`, `range permission markers`, `alternate format chunks`, `custom XML`, `structured document tags` as well as `paragraphs`. It took time to understand how to fix (repair) paragraph structures w.r.t. variation points. All of these other structures would require repairs too if they were colored.

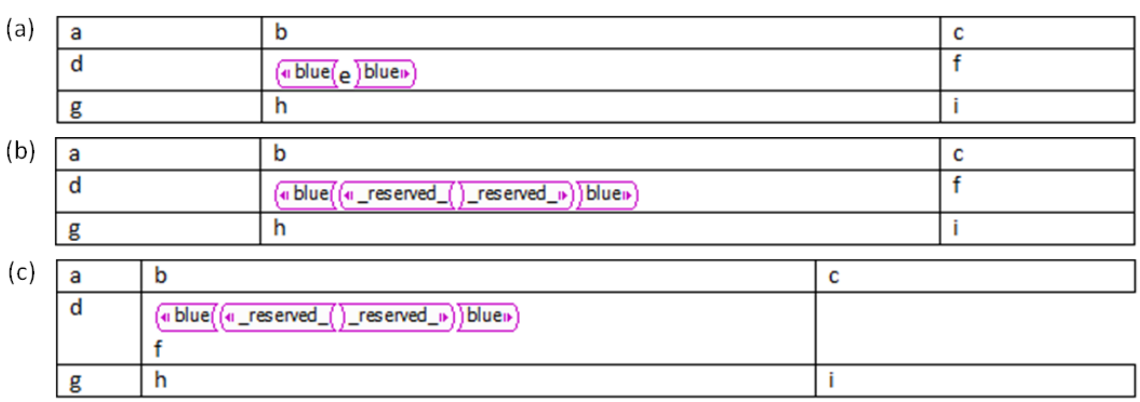


Figure 3.4: Projection Error

Figure 3.4a shows a 3×3 `table` whose mid-most cell is wrapped by `blue`. We expect Figure 3.4b as a projection where the `blue` feature was not selected, but the table that is produced is the ugly version in Figure 3.4c. Figure 3.5 shows the XML codes corresponding to Figure 3.4a. Codes inside the square say that Custom Markup tags wrap an entire cell, not the `text` of it. Accordingly, projection replaces the mid-most cell fragment with a VP, and the table loses one cell as a result.

```
<w:tbl>
    <w:tr>
        <w:tc><w:p><w:r><w:t>a</w:t></w:r></w:p></w:tc>
        <w:tc><w:p><w:r><w:t>b</w:t></w:r></w:p></w:tc>
        <w:tc><w:p><w:r><w:t>c</w:t></w:r></w:p></w:tc>
    </w:tr>
    <w:tr>
        <w:tc><w:p><w:r><w:t>d</w:t></w:r></w:p></w:tc>
        <w:customXml w:element="blue" w:uri="sampleSchema">
            <w:tc><w:p><w:r><w:t>e</w:t></w:r></w:p></w:tc>
        </w:customXml>
        <w:tc><w:p><w:r><w:t>f</w:t></w:r></w:p></w:tc>
    </w:tr>
    <w:tr>
        <w:tc><w:p><w:r><w:t>g</w:t></w:r></w:p></w:tc>
        <w:tc><w:p><w:r><w:t>h</w:t></w:r></w:p></w:tc>
        <w:tc><w:p><w:r><w:t>i</w:t></w:r></w:p></w:tc>
    </w:tr>
</w:tbl>
```

Figure 3.5: XML codes of a Projected Table

Herein lies a difficulty in leveraging MS tagging for coloring. The semantics of tagging are not necessarily the same as those of coloring. It is not easy to understand how to repair XML code in all cases. Paan does not have a complete set of solutions, and limits coloring to paragraph tagging. This raises a more basic issue: coloring is a functionality that should be part of the design of any tool like MS Word: it should not be an after-thought, or be implemented as an after-thought (as we have done).

3.2 EXPERIMENT

We evaluated Paan on two product lines: a *Notepad* application written in Java and a *Graph Product Line (GPL)* document about implementation of different graph algorithms [15]. Paan was used to pull apart `Notepad` to create variations arising from different combinations of functionalities such as 'Find', 'Print', 'Select', etc. Figure 3.6 shows a feature diagram of the `Notepad` product line: Base is a mandatory feature, and the remaining features are optional. Each feature displays an associated toolbar and menubar buttons in user interface. We found that we could color `Notepad` using only non-wrappers or only wrappers.

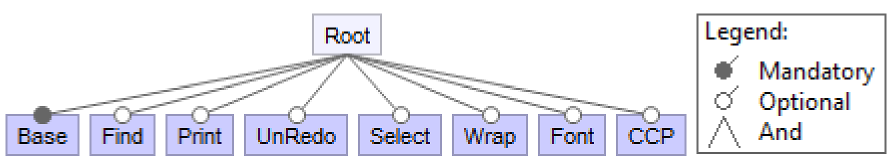


Figure 3.6: A Feature Model of Notepad

Figure 3.7a shows a declaration of JButton classes using non-wrappers. Optional features (`print` and `find`) are tagged. In Figure 3.7b, wrappers have no difference from non-wrappers as long as features do not interact. Figure 3.7c shows use of wrapper interactions. The inner-most `BASE` must exist to show optional features (`FIND` or `PRINT`).

(a) private JButton newButton, openButton, saveButton, saveAsButton, aboutButton print , printButton find , findButton ;

(b) private JButton newButton, openButton, saveButton, saveAsButton, aboutButton PRINT , printButton FIND , findButton ;

(c) private JButton FIND PRINT BASE newButton, openButton, saveButton, saveAsButton, aboutButton , printButton , findButton ;

Figure 3.7: Wrappers vs. Non-Wrappers

		Non-Wrappers/Wrappers & No Interactions	Wrappers & Interactions
Lines of Code		2074	
Available Features		25	
Possible Configurations		7056	
Tags		56	58
Depth	MAX	1	6
	AVG	1	1.51
Interactions		0	10

Table 3.1: Notepad Results

Table 3.1 makes it clear that wrappers in Figure 3.7c apparently lead to more and higher-degree feature interactions, which we found surprising. (It was our initial thought that features and feature interactions would be fundamental to a design, irrespective to whether wrappers or non-wrappers are used. Evidently, this is not the case. This raises an interesting question for future researchers: why is this so?) In any case, variations of `Notepad` can be developed incrementally by progressively exposing optional features. Figure 3.8 shows tagging features to Main class. It has a mixture of non-wrappers and wrappers.

```
public class Main extends JFrame {
   public Actions actions = new Actions(this);
   public Center center = new Center(this);
[unredo[   UndoManager undo = new UndoManager();
   UndoAction undoAction = new UndoAction(this);
   RedoAction redoAction = new RedoAction(this);]unredo]
   private JTextArea textArea;
   private JMenuBar Menubar;
   private JMenu filE, ediT, vieW, formaT, helP;
   private JMenuItem [FIND[ [SELECT[ [CCP[ [FONT[ [PRINT[ [BASE[ neW, opeN, savE, saveAS, exiT,
abouT]], prinT]], fonT]], cuT, copY, pastE]], selectALL]], finD, findNexT]];
   private JCheckBoxMenuItem lineWraP;
   private JToolBar toolBar;
   private JButton [FONT[ [FIND[ [CCP[ [UNREDO[ [PRINT[ [BASE[ newButton, openButton, saveButton,
saveAsButton, aboutButton]], printButton]], undoButton, redoButton]], cutButton, copyButton,
pasteButton]], findButton]], fontButton]];
   private JScrollPane scrollpane;
[wrap[   public JCheckBoxMenuItem getLineWrap() {
      return lineWraP;
   }]wrap]
```

Figure 3.8: Main Class of Notepad

We used three practical configurations from `Notepad`: editing, publishing, and reading. Editing has basic features to write, delete, and modify plain text along with

‘Find’ and ‘Undo/Redo’. Publishing includes ‘Print’, ‘Font’ and ‘Select’. Only for opening `Notepad` to read, ‘Wrap’ and ‘Find’ should be enough. Figure 3.9 shows these variations of `Notepad`. Figure 3.9a has all features. Figure 3.9b, 3.9c, and 3.9d are the editing, publishing and reading configurations, respectively. (Note: we produced these versions by making Word documents out of each Java file. Projected Word files were then reduced to the text of Java files, which were then compiled and run. It is from these executions that the figures below were obtained).

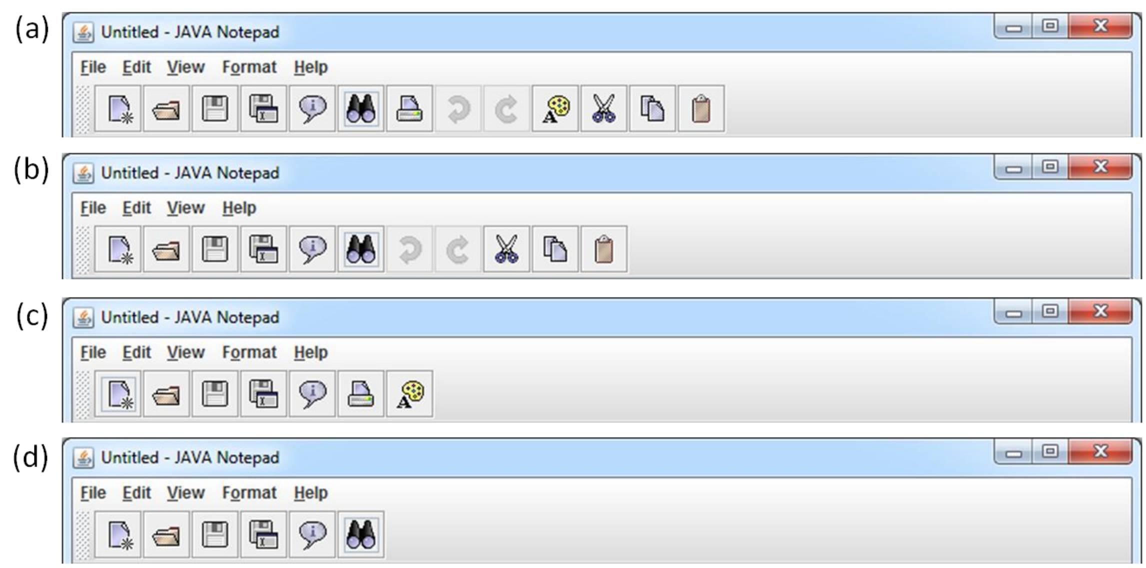


Figure 3.9: Notepad Variations

`GPL` has 1713 LOC with 18 features and 156 configurations. Its variations originate from algorithms (e.g. BFS and DFS) and structures of the graph (e.g. directed, weighted). Figure 3.10 shows a feature diagram of the `GPL` product line. Table 3.2 explains each feature briefly.

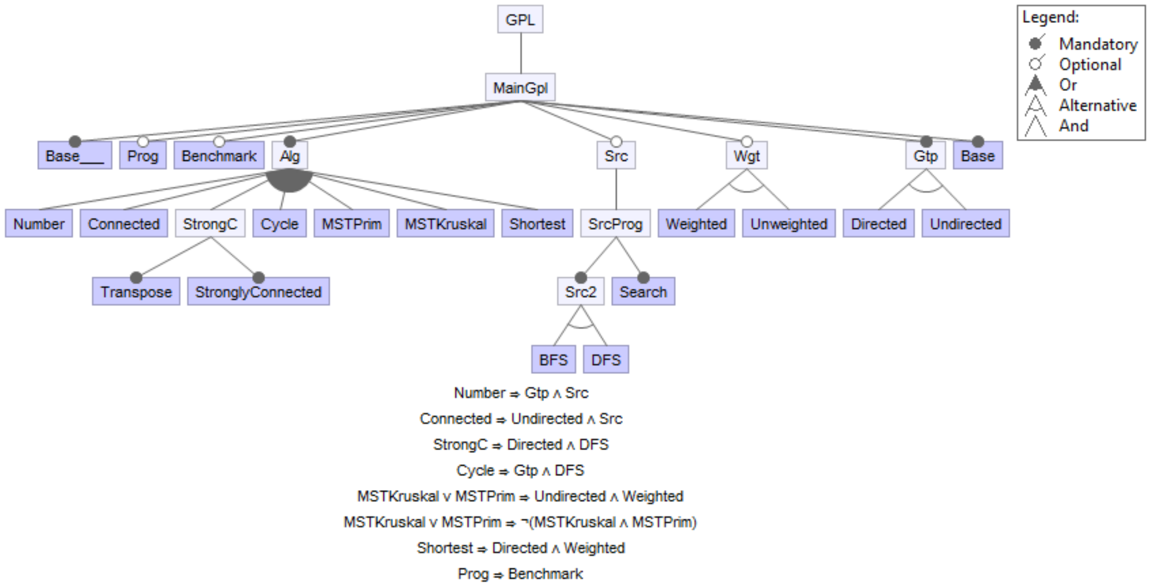


Figure 3.10: A Feature Model of GPL

Prog	Creates the objects required to represent a graph, and calls the algorithms of the family member on this graph
Benchmark	Contains functions to read a graph from a file
Vertex Numbering (Number)	Assigns a unique number to each vertex as a result of a graph traversal
Connected Components (Connected)	Computes the connected components of an undirected graph
Transpose	Graph transposition
Strongly Connected Components (StronglyConnected)	Computes the strongly connected components of a directed graph
Cycle Checking (Cycle)	Determines if there are cycles in a graph
Minimum Spanning Tree (MST Prim, MST Kruskal)	Computes a Minimum Spanning Tree (MST)
Single-Source Shortest Path (Shortest)	Computes the shortest path from a source vertex to all other vertices
Breadth First Search (BFS)	The standard breadth first search algorithm
Depth First Search (DFS)	The standard depth-first search algorithm
Weighted/Unweighted	Weighted/Unweighted graph
Directed/Undirected	Directed/Undirected graph

Table 3.2: Feature Definition in GPL

A `GPL` document in HTML format can be factored into features. It is composed of several sections: <header>, <list of links to algorithms>, <section on programmatic invocation>, <algorithm descriptions>, <implementation notes>, and <command-line invocation>. An algorithm feature, BFS for example, contributes a line to the <list of links> section and a few lines in the <algorithm description> sections. We used Paan to synthesize a Word document that is an instruction manual for each `GPL` product. Each data structure is described by a brief paragraph followed by a JPG image as shown in Figure 3.11. An instruction manual for a `GPL` product contains only one rectangular region corresponding to one of data structures: OnlyVertices, Neighbors-List, Edge-List

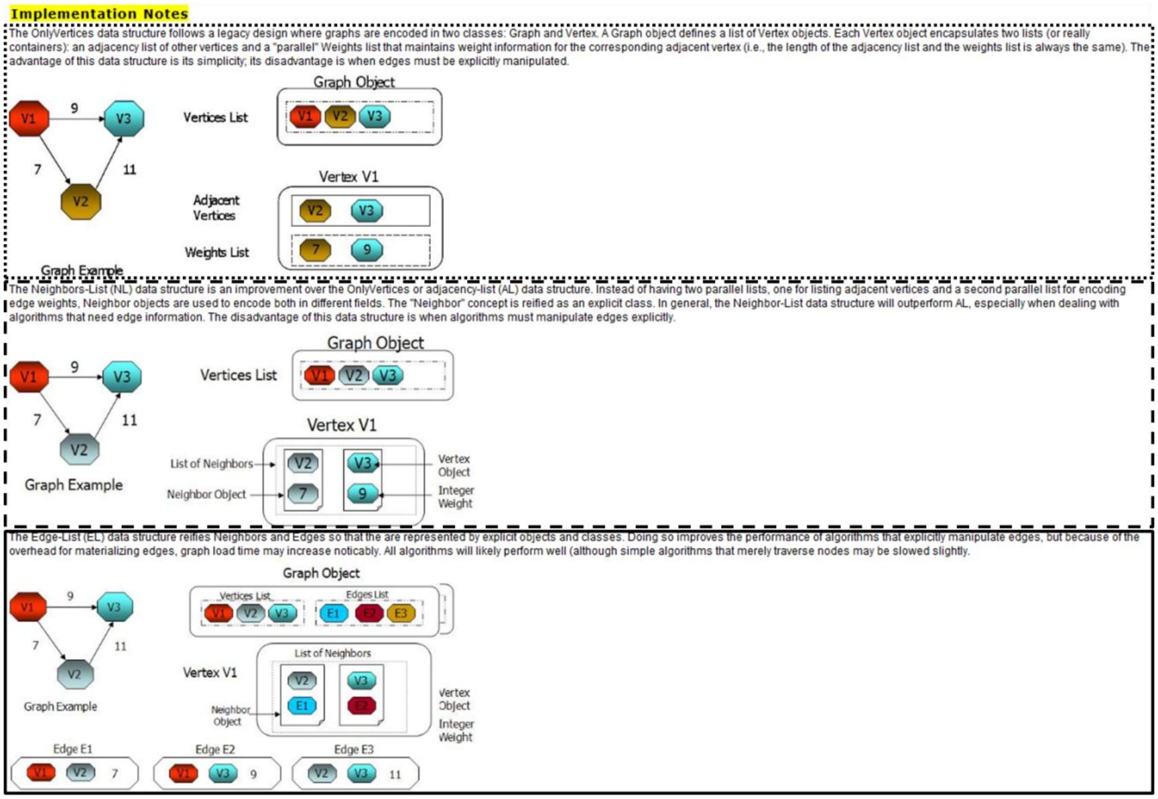
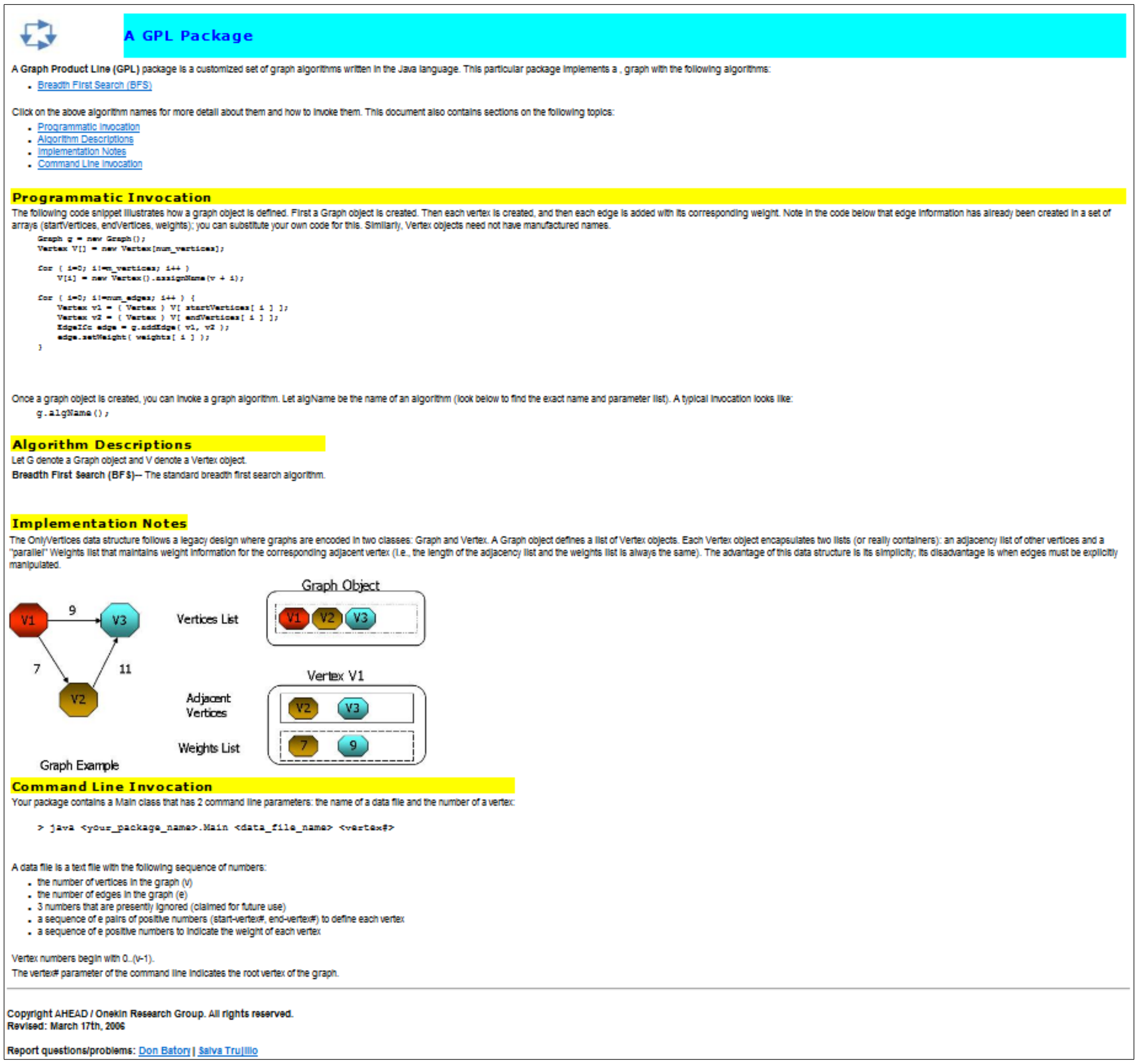


Figure 3.11: Implementation Notes of a GPL Document

Lines of Code		2074
Available Features		18
Possible Configurations		156
Tags		15
Depth	MAX	2
	AVG	2
Interactions		15

Table 3.3: GPL Document Results

Table 3.3 shows the master file with for a `GPL` document about eight algorithms, three different data structures, and a base feature, from which productions can be made. We used non-wrappers for tagging. All feature interactions arise between one optional feature and the base. Therefore, the interaction depth is always two. Figure 3.12 shows a projected version of the document where the only selected is BFS.



A GPL Package

A Graph Product Line (GPL) package is a customized set of graph algorithms written in the Java language. This particular package implements a , graph with the following algorithms:

- Breadth First Search (BFS)

Click on the above algorithm names for more detail about them and how to invoke them. This document also contains sections on the following topics:

- Programmatic Invocation
- Algorithm Descriptions
- Implementation Notes
- Command Line Invocation

Programmatic Invocation

The following code snippet illustrates how a graph object is defined. First a Graph object is created. Then each vertex is created, and then each edge is added with its corresponding weight. Note in the code below that edge information has already been created in a set of arrays (startVertices, endVertices, weights); you can substitute your own code for this. Similarly, Vertex objects need not have manufactured names.

```
Graph g = new Graph();
Vertex V[] = new Vertex[num_vertices];

for ( i=0; i!=n_vertices; i++ )
    V[i] = new Vertex().assignName(v + i);

for ( i=0; i!=num_edges; i++ ) {
    Vertex v1 = ( Vertex ) V[ startVertices[ i ] ];
    Vertex v2 = ( Vertex ) V[ endVertices[ i ] ];
    EdgeIfc edge = g.addEdge( v1, v2 );
    edge.setWeight( weights[ i ] );
}
```

Once a graph object is created, you can invoke a graph algorithm. Let algName be the name of an algorithm (look below to find the exact name and parameter list). A typical invocation looks like:

```
g.algName();
```

Algorithm Descriptions

Let G denote a Graph object and V denote a Vertex object.

Breadth First Search (BFS)-- The standard breadth first search algorithm.

Implementation Notes

The OnlyVertices data structure follows a legacy design where graphs are encoded in two classes: Graph and Vertex. A Graph object defines a list of Vertex objects. Each Vertex object encapsulates two lists (or really containers): an adjacency list of other vertices and a "parallel" Weights list that maintains weight information for the corresponding adjacent vertex (i.e., the length of the adjacency list and the weights list is always the same). The advantage of this data structure is its simplicity; its disadvantage is when edges must be explicitly manipulated.

Command Line Invocation

Your package contains a Main class that has 2 command line parameters: the name of a data file and the number of a vertex:

```
> java <your_package_name>.Main <data_file_name> <vertex#>
```

A data file is a text file with the following sequence of numbers:

- the number of vertices in the graph (v)
- the number of edges in the graph (e)
- 3 numbers that are presently ignored (claimed for future use)
- a sequence of e pairs of positive numbers (start-vertex#, end-vertex#) to define each vertex
- a sequence of e positive numbers to indicate the weight of each vertex

Vertex numbers begin with 0..(v-1).

The vertex# parameter of the command line indicates the root vertex of the graph.

Revised: March 17th, 2006

Report questions/problems: Don Batory | Salva Trujillo

Figure 3.12: A Projected GPL Document

Chapter 4: Related Work

Colored IDE (CIDE) is an advance in FOSD tooling that visualizes features and their interactions, and supports feature splitting and merging [23]. `CIDE` has preprocessor semantics, where the code of a feature is effectively surrounded by `#ifdef-#endif` statements, although it goes beyond traditional preprocessors by using ASTs rather than text. Paan differs from `CIDE` in several respects. One, obviously, is the use of MS Word documents where `CIDE` could not be used. Further, Paan relies on MS Word custom markups for coloring. Paan also differs from `CIDE` in that it supports wrappers.

Czarnecki and Antkiewicz propose an approach to map feature models to elements of UML activity diagrams using model templates [12]. UML elements are annotated with *presence conditions* (constraints similar to predicates in Paan) which are mapped from the original feature model. Using a tool called *fmp2rsm*, variants of UML models are created by removing elements (fragments) whose conditions evaluate to false. It is remarkable that `fmp2rsm` allows arbitrary propositional formulas in presence conditions whereas Paan does not permit the `NOT` operation in predicates. Moreover, `fmp2rsm` guarantees syntactic correctness in generating variants, since the generations are not performed directly on the source code but on an abstract representation like the AST used in `CIDE`.

Rabiser et al. suggest a tool-supported approach to generate product-specific documents in SPLs [32]. It uses the decision-oriented DOPLER approach for resolving variability [39]. The DOPLER tool suit adopts DocBook for variability modeling in documents [14]. The XML schema, *Document Type Definition (DTD),* is extended to define elements and attributes for implementing VPs in documents. Although the DocBook system as of computer documentation standards is suitable for automatic document processing, it is quite challenging to convert between other types of documents and DocBook. MS Word documents commonly-used in commercial domains should be converted manually. It is an apparently tedious and error-prone process.

pure::variants is a commercial variant management application supporting realization of product lines throughout the entire development phase [16]. Using Custom XML Markup, it generates variants of a MS Word document from feature configurations. However, unlike Paan, it does not provide any functionalities to update changes to the original documents.

In [13], a programming language is developed incrementally through the addition of features. In adding `Generics` to the calculus of *Featherweight Java (FJ)* to produce the calculus of *Generic Featherweight Java (GFJ)*, the required changes are woven throughout the syntax and semantics of `FJ`. The left-hand column of Figure 4.1 presents a subset of the syntax of `FJ`, the rules which formalize the subtyping relation that establish the inheritance hierarchy, and the typing rule that ensures expressions for object creation are well-formed. The corresponding definitions for `GFJ = Generics×FJ` appear in the right-hand column where shading (similar to tagging in Paan) indicates differences. These highlighted changes are the fragments of definitions that belong to the `Generics#FJ` color. However, in this work, coloring is used as a means of explanation, rather than as a tool to project colors thereby producing different variants.

FJ Expression Syntax	FJ • Generic Expression Syntax
$\texttt{e} ::= \texttt{x}$ $\mid\ \texttt{e.f}$ $\mid\ \texttt{e.m}\ (\overline{\texttt{e}})$ $\mid\ \texttt{new C}(\overline{\texttt{e}})$ $\mid\ (\texttt{C})\ \texttt{e}$	$\texttt{e} ::= \texttt{x}$ $\mid\ \texttt{e.f}$ $\mid\ \texttt{e.m}\langle\overline{\texttt{T}}\rangle\ (\overline{\texttt{e}})$ $\mid\ \texttt{new C}\langle\overline{\texttt{T}}\rangle\ (\overline{\texttt{e}})$ $\mid\ (\texttt{C}\langle\overline{\texttt{T}}\rangle)\ \texttt{e}$
FJ Subtyping $\quad$ $\texttt{T} <: \texttt{T}$	**GFJ Subtyping** $\quad$ $\Delta \vdash \texttt{T} <: \texttt{T}$
	$\Delta \vdash \texttt{X} <: \Delta(\texttt{X})$ (GS-VAR)
$\dfrac{\texttt{S<:T} \quad \texttt{T<:V}}{\texttt{S<:V}}$ (S-TRANS)	$\dfrac{\Delta \vdash \texttt{S<:T} \quad \Delta \vdash \texttt{T<:V}}{\Delta \vdash \texttt{S<:V}}$ (GS-TRANS)
$\texttt{T<:T}$ (S-REFL)	$\Delta \vdash \texttt{T<:T}$ (GS-REFL)
$\dfrac{\texttt{class C extends D}\ \{\ldots\}}{\texttt{C<:D}}$ (S-DIR)	$\dfrac{\texttt{class C}\langle\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\rangle\ \texttt{extends D}\langle\overline{\texttt{V}}\rangle\ \{\ldots\}}{\Delta \vdash \texttt{C}\langle\overline{\texttt{T}}\rangle <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{D}\langle\overline{\texttt{V}}\rangle}$ (GS-DIR)
FJ New Typing $\quad$ $\Gamma \vdash \texttt{e} : \texttt{T}$	**GFJ New Typing** $\quad$ $\Delta; \Gamma \vdash \texttt{e} : \texttt{T}$
$\dfrac{fields(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{f}} \quad \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{C}} \quad \overline{\texttt{C}} <: \overline{\texttt{D}}}{\Gamma \vdash \texttt{new C}(\overline{\texttt{e}}) : \texttt{C}}$ (T-NEW)	$\dfrac{\Delta \vdash \texttt{C}\langle\overline{\texttt{T}}\rangle \quad fields(\texttt{C}\langle\overline{\texttt{T}}\rangle) = \overline{\texttt{V}}\ \overline{\texttt{f}} \quad \Delta; \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{U}} \quad \Delta \vdash \overline{\texttt{U}} <: \overline{\texttt{V}}}{\Delta; \Gamma \vdash \texttt{new C}\langle\overline{\texttt{T}}\rangle(\overline{\texttt{e}}) : \texttt{C}}$ (GT-NEW)

Figure 4.1: Selected FJ Definitions with GFJ Changes

Chapter 5: Conclusion

The main contribution of this thesis is to extend prior work on program synthesis in product-lines. In particular, we examined projectional approaches, called coloring, where a complete document is partitioned into sections with distinct colors. Each feature is associated with a distinct color, so the removal (or projection) of that feature from the document will yield a subdocument (called a projected document) that contains only the features that are needed. The novelty of this work shows how users can edit projected documents, and these changes can be propagated back into the product line definition. Our idea is inspired in part by studies on feature interactions (i.e. changes to a feature's behavior): A document could expose only its projections due to some features encompassing proprietary or sensitive data. Therefore, changes to a projected document should be automatically propagated to its product-line definition file(s). By making the feature interactions explicit, a solution was possible.

Paan is implemented as a tool to demonstrate that back-propagation is feasible. Paan intelligently leveraged the Custom Markup to achieve coloring of MS Word documents. Paan also natively supported wrapping, a form of coloring that has different semantics of nested (`#ifdef-#endif`) preprocessor semantics. (Ultimately, Paan will be used for experiments later to evaluate the differences between nested colors and wrapping colors). However, the key novelty of Paan is its ability to shred a projected document into fragments, and update only those that are new or have changed in the tile repository, ignoring unchanged fragments.

Paan resolved some critical errors that break block-level structures inside a `body` XML structure during projection and back-propagation. However, it is not sturdy against any possible structures which are syntactically valid beyond `paragraph`, `run`, and `text`. Here in lies difficulties in understanding the complete standards of Office Open XML. Moreover, MS Word rearranges tags in order to perform code optimization.

During experiments, it found to be time-consuming and tedious job to copy codes between Word documents and other file types. We hope to improve Paan to automate conversion of MS Word documents to text representations others as well.

Bibliography

[1] A. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools Second Edition*. Pearson Education, 2006.

[2] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. Journal of Object Technology (JOT), 8(5):49--84, 2009.

[3] S. Apel, C. Kästner, and D. Batory. Aspectual feature modules. In GPCE, 2008.

[4] S. Apel, C. Kästner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In ICSE, 2009.

[5] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting dependences and interactions in feature-oriented design. In ISSRE, 2010.

[6] P. Bassett. Frame-based software engineering. IEEE Software, 4(4), 1987.

[7] D. Batory. Ahead tool suite. http://www.cs.utexas.edu/users/schwartz/ATS.html.

[8] D. Batory. Feature Models, Grammars, and Propositional Formulas. In SPLC, Sept. 2005.

[9] D. Batory, J. Kim, and P. Höfner. Understanding Feature Interactions. 2011.

[10] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM TOSEM, 1992.

[11] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. IEEE TSE, June 2004.

[12] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In R. Glück and M. R. Lowry,

editors, GPCE, volume 3676 of Lecture Notes in Computer Science, pages 422–437. Springer, 2005.

[13] B. Delaware, W. Cook, and D. Batory. Engineering modular metatheory. Submitted, 2011.

[14] Dhungana, D., Rabiser, R., Grünbacher, P., Lehner, K., Federspiel, C.: DOPLER: An Adaptable Tool Suite for Product Line Engineering. In: Proceedings of the 11th International Software Product Line Conference (SPLC 2007), Kyoto, Japan, vol. 2, pp. 151–152. Kindai Kagaku Sha Co. Ltd. (2007).

[15] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2002.

[16] Pure:systems GmbH: Automatic Generation of Word Document Variants (2010), http://www.pure-systems.com/flash/pv-wordintegration/flash.html.

[17] ECMA International, "Office Open XML File Formats", 1st Edition, ECMA-376, December 2006.

[18] ECMA International, "Office Open XML File Formats", 2nd Edition, ECMA-376, December 2008.

[19] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. IEEE TSE, October 1998.

[20] K. Kang. Private Correspondence, Oct. 2003.

[21] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. CMU/SEI-90-TR-021, 1990.

[22] C. Kästner, S. Apel, and D. Batory. A case study implementing features using aspect. In SPLC, 2007.

[23] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In ICSE, 2008.

[24] C. Kästner end et al. On the impact of the optional feature problem: Analysis and case studies. In SPLC, 2009.

[25] C. H. P. Kim, C. Kästner, and D. Batory. On the modularity of feature interactions. In GPCE, 2008.

[26] G. Kiczales et al. Aspect-Oriented Programming. In ECOOP, 1997.

[27] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In ICSE, 2006.

[28] J. Liu, D. Batory, and S. Nedunuri. Modeling interactions in feature oriented designs. *In Int. Conf. on Feature Interactions*, 2005.

[29] R. E. Lopez-herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In Proc. 2001 Conf. Generative and Component-Based Software Eng, pages 10-24. Springer, 2001.

[30] C. Prehofer. Feature Oriented Programming: A Fresh Look at Objects. In ECOOP, 1997.

[31] Feature interaction problem. http://en.wikipedia.org/wiki/Feature_interaction_problem.

[32] R. Rabiser, W. Heider, C. Elsner, M. Lehofer, P. Grünbacher, and C. Schwanninger. A flexible approach for generating product-specific documents in product lines. In Proceedings of the 14th International Software Product Line Conference, pages 47-61, Jeju Island, South Korea, 2010. Springer-Verlag Berlin Heidelberg.

[33] D. Roundy. Darcs: Distributed version management in Haskell. In ACM SIGPLAN Workshop on Haskell, 2005.

[34] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In SPLC, 2010.

[35] S. Sobernig. Feature interaction networks. In ACM *Symposium on Applied Computing*, 2010.

[36] Stayton, B.: DocBook XSL. Sagehill Enterprises (2005).

[37] International Organization for Standardization, "Information technology -- Document description and processing languages -- Office Open XML File Formats -- Part 1: Fundamentals and Markup Language Reference -- Amendment 1", ISO/IEC 29500-1:2008/Amd.1:2010.

[38] International Organization for Standardization, "Information technology -- Document description and processing languages -- Office Open XML File Formats -- Part 2: Open Packaging Conventions", ISO/IEC 29500-2:2008.

[39] International Organization for Standardization, "Information technology -- Document description and processing languages -- Office Open XML File Formats -- Part 3: Markup Compatibility and Extensibility", ISO/IEC 29500-3:2008.

[40] International Organization for Standardization, "Information technology -- Document description and processing languages -- Office Open XML File Formats -- Part 4: Transitional Migration Features -- Amendment 1", ISO/IEC 29500-4:2008/Amd.1:2010.

[41] Office Open XML, http://en.wikipedia.org/wiki/Office_Open_XML.

Vita

Jongwook Kim was born in Republic of Korea on March 27th, 1980, the son of Yongwoon Kim and Choonja Kim. He received the degree of Bachelor of Engineering in Computer Science and Engineering from Korea University in 2008. After the graduation, he entered the Graduate School at the University of Texas at Austin.

Permanent address: 7600 Wood Hollow DR APT 103, Austin, TX 78731

This thesis was typed by the author.