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 .............................................................................................................. ix
List of Figures .............................................................................................................. x

Chapter 1:  Introduction ................................................................................................. 1
  1.1 Painting Programs ................................................................................................. 3
  1.2 Tiles and Projection .............................................................................................. 4
  1.3 Tile Implementations and Variation Points ......................................................... 4
  1.4 Back-Propagation of Changes ............................................................................. 6

Chapter 2:  Design ........................................................................................................ 7
  2.1 Office Open XML ................................................................................................ 7
  2.2 Tagging Features ................................................................................................ 7
  2.3 Nested Preprocessor Semantics ......................................................................... 8
  2.4 Wrapping and Wrappers .................................................................................... 9
  2.5 N-Way Interactions ............................................................................................. 11
  2.6 Projection ........................................................................................................... 14
    2.6.1 File Projection ............................................................................................... 14
    2.6.2 Tag Projection .............................................................................................. 14
  2.7 Back-Propagation ............................................................................................... 16
  2.8 Merging Tiles ..................................................................................................... 17

Chapter 3:  Evaluation .................................................................................................. 21
  3.1 Experience .......................................................................................................... 21
  3.2 Experiment .......................................................................................................... 25

Chapter 4:  Related Work ............................................................................................ 34

Chapter 5:  Conclusion ................................................................................................ 37

Bibliography .................................................................................................................. 39
Vita ............................................................................................................................... 43
List of Tables

Table 2.1: Tile Mergence Rules ...........................................................................18
Table 3.1: Notepad Results ..................................................................................26
Table 3.2: Feature Definition in GPL .................................................................31
Table 3.3: GPL Document Results.......................................................................32
List of Figures

Figure 1.1: The Counted Stack .................................................................3
Figure 1.2: The Counted Stack with Variation Points.........................5
Figure 2.1: MS Word Custom Markup Tags and its XML ..................8
Figure 2.2: Nesting and Projection of Tags ..........................................9
Figure 2.3: Variants of Wrappers ..........................................................10
Figure 2.4: Wrappers ...........................................................................11
Figure 2.5: Non-Wrapper Interaction and Predicate .........................12
Figure 2.6: Wrapper Interaction and Predicate ....................................12
Figure 2.7: Tile Interaction and Predicate ............................................13
Figure 2.8: Projecting Wrappers ..........................................................15
Figure 2.9: Multiple Variation Points of Single Wrapper .................16
Figure 2.10: Tile Merging .................................................................18
Figure 2.11: Tile Mergence for Inner Non-Wrappers .....................19
Figure 2.12: Tile Mergence of Wrappers ...........................................19
Figure 2.13: Invalid Tile Mergence ......................................................20
Figure 3.1: Back-Propagation Error ....................................................21
Figure 3.2: Block-Level Structures in XML Codes .........................22
Figure 3.3: Block-Level Error Correction ............................................23
Figure 3.4: Projection Error ...............................................................23
Figure 3.5: XML codes of a Projected Table ...................................24
Figure 3.6: A Feature Model of Notepad ...........................................25
Figure 3.7: Wrappers vs. Non-Wrappers ..........................................26
Figure 3.8: Main Class of Notepad .....................................................28
Figure 3.9: Notepad Variations .................................................................29
Figure 3.10: A Feature Model of GPL .......................................................30
Figure 3.11: Implementation Notes of a GPL Document .........................32
Figure 3.12: A Projected GPL Document ................................................33
Figure 4.1: Selected FJ Definitions with GFJ Changes ...........................36
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.

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

    String s = new String();
    void empty() {
        ctr = 0;
        s = "";
    }
    void push(char a) {
        ctr++;
        s = s + String.valueOf(a) + 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:

$$\text{Doc} = \text{BASE#STACK#COUNT} \cdot \text{BASE#COUNT} \cdot \text{BASE#STACK} \cdot \text{BASE}$$

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:

$$\text{DocNoC} = \text{BASE#STACK} \cdot \text{BASE}$$

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, it 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 \( \Delta P \). 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:

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

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.

![XML representation of a fragment surrounded by blue tags](image)

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.

![Diagram showing tagging and projection](image)

**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.
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).
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 $\land$-operation in conjunctive normal form (CNF).
Figure 2.5: Non-Wrapper Interaction and Predicate

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.

![Diagram](image)

**Figure 2.7: Tile Interaction and Predicate**

<table>
<thead>
<tr>
<th>Tile</th>
<th>Predicate</th>
<th>Base-Wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$A \land (D \lor F \lor H)$</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>$B \land (D \lor F)$</td>
<td>No</td>
</tr>
<tr>
<td>c</td>
<td>$B \land (D \lor F) \land c$</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>Yes</td>
</tr>
<tr>
<td>e</td>
<td>$D \land e$</td>
<td>-</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>Yes</td>
</tr>
<tr>
<td>g</td>
<td>$A \land g$</td>
<td>-</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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 \( \mathcal{W} \) in its repository in its entirety. Let \( \mathcal{P} \) denote the set of features that were selected,
meaning that their fragments are to remain after projection. A traversal of \( \mathcal{W} \) 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 \( \mathcal{W}_p \). 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](image-url)
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.

![Image](attachment:figure2.png)

Figure 2.9: Multiple Variation Points of Single Wrapper

Let \( \hat{W} \) be a tagged MS Word document and let \( \hat{W}_p \) be a projection of \( \hat{W} \), where \( p \) is a set of features. Therefore, \( \hat{W}_p \) eliminates all fragments from \( \hat{W} \) whose set of colors do not belong to \( p \). A user can now modify \( \hat{W}_p \) 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 \( \hat{W}_p \) to \( \hat{W} \), Paan maintains a copy of \( \hat{W} \) in its repository that existed prior to projection. It then traverses \( \hat{W}_p \) to locate VPs whose
fragments have been projected. For each such VP \( i \), it finds fragment \( i \) in \( \bar{W} \) and restores that fragment in \( \bar{W}_p \). At the end, all projected fragments in \( \bar{W}_p \) have been restored with their original contents. Paan then discards the original copy \( \bar{W} \) and replaces it with \( \bar{W}_p \). And the projection-back-propagate cycle continues. Here, the restoration of projected VPs can be accomplished in linear time, since a single pass through \( \bar{W} \) is enough to find all (VP, fragment) pairs and a single pass through \( \bar{W}_p \) 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 \( \bar{W}_p \) have been modified, 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.
During projection, redundant tags are removed since their predicates are always true. Tile merging is applied to wrappers as well in the following rules:

<table>
<thead>
<tr>
<th>Outer</th>
<th>Inner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Wrapper</td>
<td>Non-Wrapper</td>
</tr>
<tr>
<td>Wrapper</td>
<td>Implicative</td>
</tr>
</tbody>
</table>

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.
Figure 2.11: Tile Mergence for Inner Non-Wrappers

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.

![Diagram](image)

**Figure 2.13: Invalid Tile Mergence**

<table>
<thead>
<tr>
<th>Tile</th>
<th>Predicate</th>
<th>Base-Wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>GREEN</td>
<td>GREEN(\land)BLUE</td>
<td>No</td>
</tr>
<tr>
<td>red</td>
<td>GREEN(\land)BLUE(\land)red</td>
<td>-</td>
</tr>
<tr>
<td>RED</td>
<td>RED(\land)BLUE</td>
<td>No</td>
</tr>
<tr>
<td>BLUE</td>
<td>BLUE</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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.

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.

Figure 3.2: Block-Level Structures in XML Codes
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.

![XML Codes of a Projected Table](image)

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.

![Feature Diagram of Notepad](image)

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).
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.
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).

![Notepad Variations](image)

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

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.
Table 3.3: GPL Document Results

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>2074</td>
</tr>
<tr>
<td>Available Features</td>
<td>18</td>
</tr>
<tr>
<td>Possible Configurations</td>
<td>156</td>
</tr>
<tr>
<td>Tags</td>
<td>15</td>
</tr>
<tr>
<td>Depth</td>
<td></td>
</tr>
<tr>
<td>MAX</td>
<td>2</td>
</tr>
<tr>
<td>AVG</td>
<td>2</td>
</tr>
<tr>
<td>Interactions</td>
<td>15</td>
</tr>
</tbody>
</table>

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.

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.
<table>
<thead>
<tr>
<th><strong>FJ Expression Syntax</strong></th>
<th><strong>FJ • Generic Expression Syntax</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( e ::= x )</td>
<td>( e ::= x )</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
</tbody>
</table>

**FJ Subtyping**  \( T <: T \)  **GFJ Subtyping**  \( \Delta \vdash T <: T \)

\[
\frac{S <: T \quad T <: V}{S <: V} \quad \text{(S-TRANS)}
\]

\[
\frac{T <: T}{(S-REFL)}
\]

\[
\frac{\text{class } C \text{ extends } D \{ \ldots \}}{C <: D} \quad \text{(S-DIR)}
\]

**FJ New Typing**  \( \Gamma \vdash e : T \)  **GFJ New Typing**  \( \Delta ; \Gamma \vdash e : T \)

\[
\frac{\text{fields}(C) = D \{ \bar{Y} \} \quad \Gamma \vdash \bar{a} : C \quad C <: D}{\Gamma \vdash \text{new } C \langle \bar{u} \rangle : C} \quad \text{(T-NEW)}
\]

\[
\frac{\Delta \vdash C \langle \pi \rangle \quad \text{fields}(C \langle \pi \rangle) = \bar{Y} \{ \bar{X} \}}{\Delta ; \Gamma \vdash \bar{a} : \bar{U} \quad \Delta \vdash \bar{U} <: \bar{V}}
\]

\[
\frac{\Delta ; \Gamma \vdash \text{new } C \langle \pi \rangle \langle \bar{e} \rangle : C}{(GT-NEW)}
\]

**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


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.