An XVCL Approach to Handling Variants: A KWIC Product Line Example

Hongyu Zhang and Stan Jarzabek
Department of Computer Science, School of Computing
National University of Singapore
Lower Kent Ridge Road, Singapore 117543
{zhanghy, stan}@comp.nus.edu.sg

Abstract

We developed XVCL (XML-based Variant Configuration Language), a method and tool for product lines, to facilitate handling variants in reusable software assets (such as architecture, code components or UML models). XVCL is a newer version of Bassett's frames [1], a technology that has achieved substantial productivity improvements in large data processing product lines written in COBOL. Despite its simplicity, XVCL can effectively manage a wide range of product line variants from a compact base of meta-components, structured for effective reuse. We applied XVCL in two medium-size product line projects and a number of smaller case studies. In this paper, we communicate XVCL's capabilities to support product lines by means of a simple, but still interesting, example of the KWIC system introduced by Parnas in 1970's. We show how we can handle functional variants, variant design decisions and implementation-level variants in a generic KWIC system.

1. Introduction

In recent years, software product line approach, initiated by Parnas in 1970s [11], has emerged as a promising approach to improve software productivity and quality. A product line (also called product family or system family) arises in situations when we develop and maintain multiple versions of essentially the same product for different customers. Apart from similarities, members of a product line also differ in some functional and non-functional requirements. A product line often emerges from a single system over years of evolution.

In the product line approach, we identify common and variant requirements, and build generic and adaptable assets such as a product line architecture, generic components, domain models, etc. For developing specific products, we reuse product line assets instead of working from scratch. Variability is inherent in the real world and stems from sources such as customer's needs, mutability of the environment or system evolution. Consequently, product line variants may be related to functional requirements, quality attributes and design/implementation decisions. A challenge in supporting a product line is to effectively handle these variants.

We developed XVCL (XML-based Variant Configuration Language), a method and tool for handling variants in reusable software assets. XVCL is a newer version of Bassett's frames [1], a technology that has achieved substantial productivity improvements in large data processing COBOL systems. An independent assessment showed that frame technology has reduced large software project costs by over 84% and their times-to-market by 70%, when compared to industry norms [1]. These gains are due to the flexibility of the resulting architectures and their evolvability over time. The record of frame technology in large-scale software applications was the main reason that led us to implement XVCL.

Using XVCL, we package program components into generic, adaptable meta-components, called x-frames. X-frames contain actual code of program components, instrumented with XVCL commands for change. A hierarchy of inter-related x-frames forms an x-framework. Concrete programs are produced from an x-framework, using "composition with adaptation" mechanism. During x-frame processing, the XVCL processor traverses an x-framework, performs composition and adaptation by executing XVCL commands embedded in x-frames, and produces components for a specific system, a member of the product line.

We developed and experimented with XVCL in a collaborative project involving two universities and two industrial partners in Singapore and Canada[1]. We applied XVCL in two medium-size product line projects and a number of smaller case studies. These experiments showed that XVCL blends with contemporary programming paradigms, offering an effective reuse mechanism on top of other mechanisms supported by those paradigms. In our recent case study, we showed that the Java Buffer library can be produced from x-frames

1 Project funded by Singapore National Science and Technology Board and Canadian Ministry of Energy, Science and Technology, involving National University of Singapore, SES Systems Pte Ltd, University of Waterloo and Netron, Inc. (Toronto).

containing 32% of code that we find in the original Java classes [7].

In previous papers, we described implementation [15] and basic usage of XVCL [14], our experimentation in CAD product line [16] and application of XVCL to handle variants in UML models [8]. In this paper, we describe how we used XVCL to handle functional variants, variant design decisions and implementation-level variants in a generic KWIC system (Key Word in Context). KWIC system is a well-known example introduced by Parnas in 1970's [10].

In September 2002, we made XVCL available at Open Source forum (fxvcl.sourceforge.net). We hope this paper will encourage others to experiment with XVCL concepts and further explore potentials of the "composition with adaptation" techniques in software engineering. The KWIC problem is well-known to software engineering community, and can serve as a benchmark for comparing product line methods.

2. Related work

As early as in 1970's, Parnas [10] proposed to apply modularization and information hiding principles to handle variants in product lines. Object-Oriented frameworks [9] use information hiding along with inheritance, dynamic binding and design patterns [3] to handle variants in product lines. The vision of component-based software engineering is also to support product lines. An application generator approach [13] is a powerful and cost-effective solution to the product line in well understood and stable domains. Generators transform problem specifications written in domain-specific languages into concrete programs. Template meta-programming [2] also addresses certain types of variants at the program construction time. Templates are heavily exploited in the C++ Standard Template Library (STL). Templates are tightly coupled with specific programming languages such as C++.

Recent work focuses on advanced separation of concerns. Concerns in multiple dimensions may spread throughout the whole program and cannot be nicely confined to a small number of components. A number of approaches have been proposed to address crosscutting concerns and concern compositions. In aspect-oriented programming [6], each computational aspect is programmed separately and can be composed to form a complete system.

Other approaches to handling variants include macros, table-driven techniques, configuration management. It is beyond the scope of this paper to discuss these approaches in detail. We refer the reader to [2] for an excellent comparative study of various approaches to handling product families, both at construction-time and runtime.

XVCL is a meta-programming technique based on the "composition with adaptation". Like macros, x-frames contain code fragments along with adaptation commands. Unlike macros, XVCL has unique capabilities that enhance reuse, such as scoping and overriding rules for meta-variables, insertions at breakpoints and while loops executed at meta-level. Unlike OO and template techniques, XVCL can handle variants at any granularity level and can trace the impact of variants across the whole program. Unlike AoP, in XVCL we explicitly mark the points affected by variants and specify required adaptations. Unlike other methods, XVCL provides a uniform variability mechanism which can handle variants within and across a range of product line assets such as domain model, program, documentation, etc.

3. An overview of the KWIC domain

The KWIC (Key Word in Context) problem was first proposed by Parnas to contrast different criteria for modular software decomposition [10]. Since its introduction, the problem has been used in several studies to illustrate the benefits of different architectural styles [4] [12].

Parnas formulated the KWIC problem as follows:

"The KWIC [Key Word in Context] index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a list of all circular shifts of all lines in alphabetical order."

All the KWIC systems are similar in the sense that they satisfy the basic requirements as stated above. However, there are also differences across KWIC systems related to functional requirements, design decisions and implementation details. These differences (variants) yield a KWIC product line. In the remaining part of this section, we describe the variants, giving each variant a unique name for future reference.

As Parnas points out, the KWIC systems are small systems (except under extreme circumstances such as huge data base). However, we use this well-known example to illustrate principles of our approach, treating this problem as if it were a large project.

Functional variants

- NOISE_WORDS: In some KWIC systems, circular shifts that start with noise words (such as *a, the, and*, etc.) should be eliminated. Noise words may differ from one KWIC system to another.
- INPUT_METHOD: Lines can be read from a file, or from system console. Values FILE and CONSOLE denote these two options.

- OUTPUT_METHOD: Indexed lines can be output to a file, or to system console. Values FILE and CONSOLE denote these two options.
- CASE_SENSITIVE: Case sensitivity may or may not be taken into account during sorting lines. Values SENSITIVE and INSENSITIVE denote these two options.

Variant design decisions

Some variants in KWIC domain that affect architectural design are:

- SHIFT_PROCESSING: Line shifting can be performed on each line as it is read from the input device or after all the lines have been read. Values EACHLINE and ALLLINES denote these two options.
- SHIFT_DATA: Circular shifts can be stored explicitly (as a set of strings) or implicitly (as pairs of index and offset). Values EXPLICIT and IMPLICIT denote these two options.

Implementation-level variants

- PACKAGE: KWIC components for different systems may belong to different Java packages.
- NEW_ATTRIBUTES: This variant caters for the situations when new, unexpected changes in requirements require us to add more attributes into KWIC classes.
- NEW_METHODS: This variant caters for the situations when new, unexpected changes in requirements require us to add more methods into KWIC classes.
- NEW_IMPORTS: This variant caters for the situations when new, unexpected changes in requirements require us to include more Java packages into a KWIC system.

Feature diagrams [5] are often used to model common and variant product line requirements. Figure 1 shows a fragment of feature diagram for KWIC product line (see [2] for a detailed description of feature diagram).

4. The impact of variants on KWIC components

We adopt the Abstract Data Type (Object-Oriented) architectural style [10][12] for KWIC product line. In the remaining part of this section, we will consider the functional variants and variant design decisions we have identified in Section 3, and evaluate the impact of variants on KWIC components. In the following section, we will design a KWIC product line architecture to accommodate those variants.

Figure 2 depicts KWIC components as an UML component diagram. We use UML extension mechanism (tagged values) to model the impact of variants on KWIC architecture.

To accommodate the INPUT_METHOD and OUTPUT_METHOD variants, we need only modify the Input and Output component, respectively.

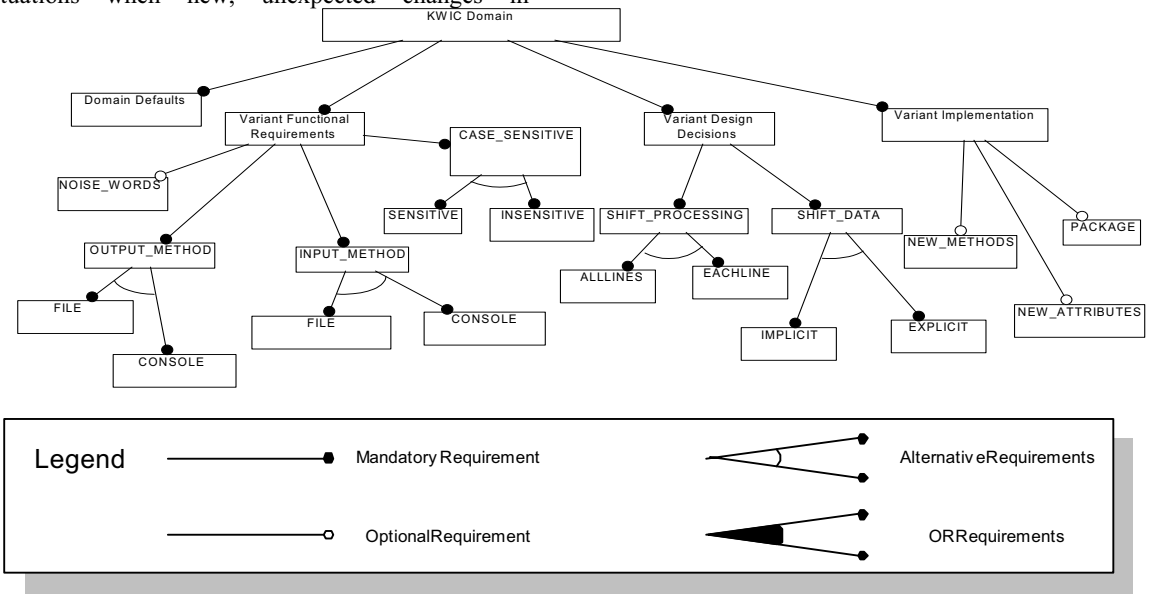

Figure 1. A feature diagram for KWIC product line (partial)

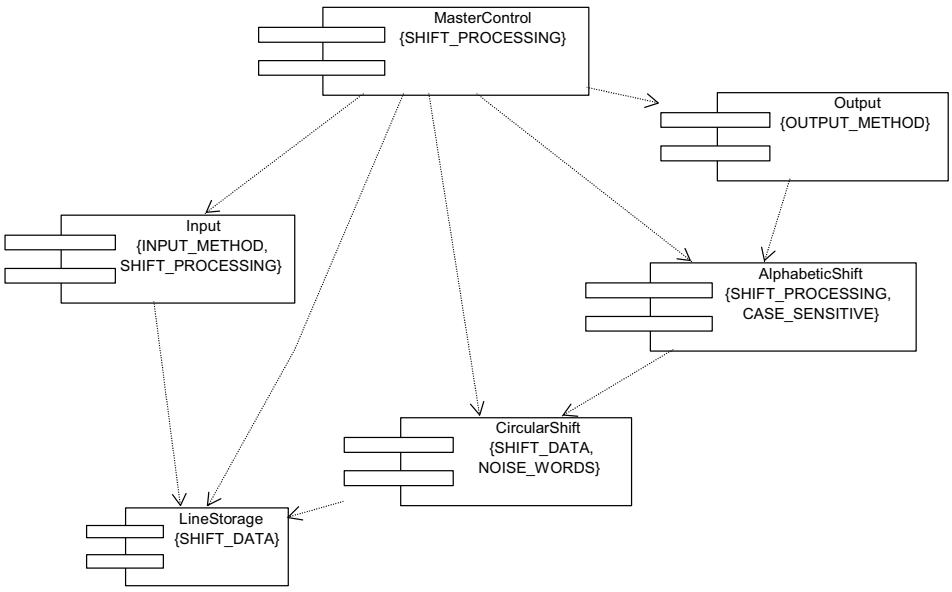

Figure 2. The KWIC product line architecture – a component view

To accommodate the SHIFT_PROCESSING variant, we need to change the Input component so that it returns control after reading one line or all lines. The MasterControl component must be changed because the number of times the CircualrShift component is invoked is different. The AlphabeticShift must also be changed either to sort lines incrementally as they are read (e.g., using, the insertion sort), or to sort all the lines at once (e.g., using the bubble sort). The CircularShift encapsulates the implementation of line shifting and provides interfaces for accessing shifted lines. Thus the CircularShift component does not need to be changed.

To accommodate the SHIFT_DATA variant, we need change the data representation of the circular shifts which affects the CircularShift and LineStorage components.

To accommodate the NOISE_WORDS variant, we need only change the CircularShift component. Other components are not affected as the CircularShift component hides the implementation of circular shifts.

To accommodate the CASE_SENSITIVE variant, we need only change the AlphabeticShift component. Other components are not affected as the AlphabeticShift hides the implementation of sorting circular shifts.

From the above discussion, we can see that one variant may affect many components, and one component may be affected by many variants.

5. An overview of a KWIC x-framework

A KWIC x-framework is an XVCL implementation of the concept of a product line architecture. An x-framework is a layered collection of meta-components that in XVCL we call x-frames. Figure 3 shows a KWIC x-framework which facilitates production of concrete KWIC components that meet required variants. KWIC x-frames are generic, in the sense that they include provisions for accommodating variants. We have six x-frames in Layer 1 in Figure 3, namely x_MasterControl, x_Input, x_LineStorage, x_CircularShift, x_AlphabeticShift and x_Output, corresponding to six KWIC components depicted in Figure 2.

X-frames at Layer 2 are building blocks of Layer 1 x-frames. At program construction time, Layer 2 x-frames are composed into Layer 1 x-frames after possible adaptation. Specifications for building a specific KWIC system are defined in the root x-frame called specification x-frame (SPC). The SPC specifies the composition and adaptation of lower-level x-frames, and records the configuration of required variants. Given SPC, the XVCL processor traverses the x-framework, performs composition and adaptation, and constructs a custom KWIC system meeting required variants.

We find that Input and Output components have much in common – when the input and output method is FILE, these two components perform similar functions: open a file, read/write lines, and then close the file. Furthermore, file accessing and the system console accessing are also similar – system console is a standard input/output device that can be treated as a special file. Thus we can design an x-frame called FileIO to capture the commonalities between Input and Output components, and between file accessing and system console accessing. Specific Input/Output components that satisfy different INPUT_METHOD/OUTPUT_METHOD variants can be generated from the FileIO x-frame after adaptation.

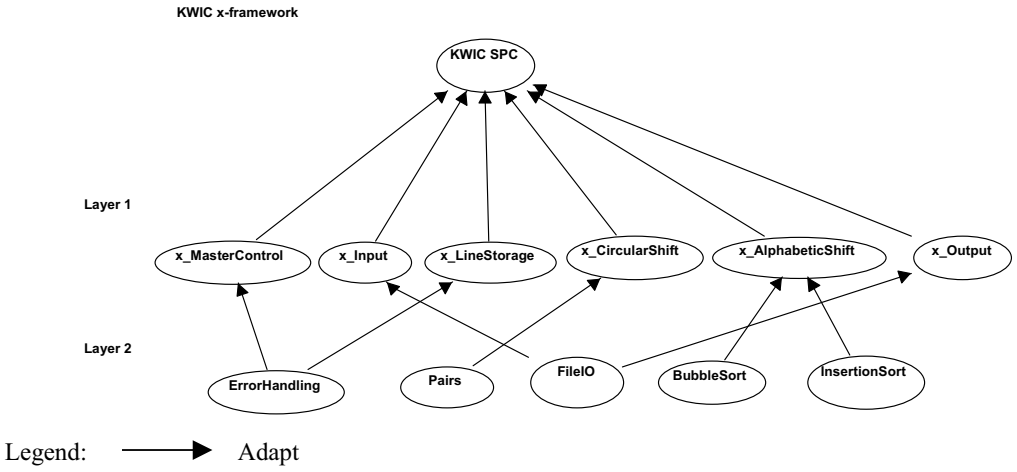

Figure 3. KWIC x-framework

The purpose of the x_AlphabeticShift x-frame is to accommodate different sorting methods into the AlphabeticShift component. In our example, we cater for two sorting methods, namely bubble and insertion sorting. These sorting methods are defined in x-frames BubbleSort and InsertionSort, respectively. These sorting x-frames contain generic method for sorting elements of any data type. Thus they can be reused for sorting in other applications as well. The x_AlphabeticShift x-frame also contains variation points to accommodate the SHIFT_PROCESSING variant and the CASE_SENSITIVE variant.

The purpose of the x_CircularShift x-frame is to accommodate variants into the CircularShift component. When we need an IMPLICIT circular shift data representation, the x_CircularShift x-frame adapts the Pairs x-frame which deals with representing circular shifts as a pair of index and offset. The x_CircularShift x-frame also contains variation points to accommodate the SHIFT_DATA and NOISE_WORDS variants.

The ErrorHandling x-frame contains code for handling errors. It is adapted by the x_MasterControl and x_LineStorage x-frames to capture the exceptions.

6. KWIC meta-components as x-frames

In this section, we show how we apply XVCL to accommodate variants into KWIC components. Due to space constraints, we only discuss the x_CircularShift x-frame for creating the CircularShift component. The reader can find the complete KWIC example at http://fxvcl.sourceforge.net. Some major XVCL commands are given in the Appendix.

The x_CircularShift x-frame (Figure 4) contains Java code instrumented with XVCL commands for change. Bold lines in Figure 4 highlight the XVCL commands.

```
<x-frame name="x_CircularShift" language="java">

<set var="PACKAGE", value="kwic"/>
<set-multi var="NOISE_WORDS" value="a, an"/>
<break name="CIRCULARSHIFT_NEW_PARAMETERS"/>

package <value-of expr="?@PACKAGE?"/>;       // Java Package
import java.util.*;
<break name="CIRCULARSHIFT_NEW_IMPORTS"/>

class CircularShift {
    Vector circularShifts;                  // Storage containing lines shifted

<break name="CIRCULARSHIFT_NEW_ATTRIBUTES"/>

    public CircularShift() {
        circularShifts = new Vector();
    }
    public Vector getShiftedLines() {
       return circularShifts;
    }
    public void circularShiftLine(String currentLine, int LineNumber) {
        StringTokenizer parser = new StringTokenizer(currentLine);
```

```
        int lastPosition = 0;
        while(parser.hasMoreTokens()) {
             String token = parser.nextToken();

      // Handle noise words. If noise words, skip processing
    <while using-items-in="NOISE_WORDS">
             if (token.equals("<value-of expr="?@NOISE_WORDS?"/>"))
                 continue;
    </while>
             lastPosition = currentLine.indexOf(token);  // Index in original lines

    <select option="SHIFT_DATA">
        <option value="EXPLICIT">            // Store Circular shifts as string data
             String circularShiftedline = currentLine.substring(lastPosition) + " " +
currentLine.substring(0, lastPosition);
             circularShifts.addElement(circularShiftedline);
        </option>
        <option value="IMPLICIT">            // Store Circular shifts as indexes and offsets
             circularShifts.addElement(new Pairs(lastPosition, LineNumber));
        </option>
    </select>
        }
       return;
    }
    …
<break name="CIRCULARSHIFT_NEW_METHODS"/>

<select option="SHIFT_DATA">
  <option value="IMPLICIT">
      <adapt x-frame = ”Pairs.xvcl”/>
  </option>
</select>
}
</x-frame>
```

Figure 4. The x_CircularShift x-frame

The x_CircularShift x-frame accommodates the following variants: NOISE_WORDS (a functional variant), SHIFT_DATA (variant design decision), PACKAGE, NEW_IMPORTS, NEW_ATTRIBUTES and NEW_METHODS (implementation-level variants). We shall now explain how these variants are addressed in Figure 4 in detail.

Accommodating the NOISE_WORDS variant

This variant requires us to eliminate lines that start with a noise word. To handle the NOISE_WORDS variant, we first define a multi-valued meta-variable NOISE_WORDS (in a `<set-multi>` command), and then use a `<while>` command to process each of its values (Figure 5).

The meta-variable NOISE_WORDS is given default value of “a, an”. This value takes effect unless some higher-level x-frames (e.g., SPC) override the default. The `<while>` command iterates over its body until all the values of meta-variable NOISE_WORDS have been processed. In each iteration, the XVCL processor builds into the component CircularShift the necessary code to handle a specific noise word.

```
   …
   <set-multi var="NOISE_WORDS" value="a, an”/>
   …
   public void circularShiftLine(String
currentLine, int LineNumber) {
   …
   <while using-items-in="NOISE_WORDS">
       if (token.equals("<value-of
expr="?@NOISE_WORDS?"/>"))
           continue;
   </while>
   …
   }
```

Figure 5. Addressing the NOISE_WORDS variant

Accommodating the SHIFT_DATA variant

The circularly shifted lines can be stored explicitly (as a set of strings) or implicitly (as pairs of index and offsets). We use `<select>` commands to accommodate this variant, as shown in Figure 6.

```
   …
   <set var="SHIFT_DATA" value="EXPLICIT"/>
   …
   public void circularShiftLine(String
currentLine, int LineNumber) {
       …
     <select option="SHIFT_DATA">
        <option value="EXPLICIT">
// Code about storing circular shifts explicitly
          …
        </option>
        <option value="IMPLICIT">
// Code about storing circular shifts implicitly
          …
        </option>
     </select>
   …
```

```
   }
    …
<select option="SHIFT_DATA">
  <option value="IMPLICIT">
      <adapt x-frame = "Pairs.xvcl"/>
  </option>
</select>
```

Figure 6. Addressing the SHIFT_DATA variant

The <set> command defines meta-variable SHIFT_DATA with default value "EXPLICIT". This value takes effect unless some higher-level x-frames (e.g., SPC) override the default. The first <select> command selects appropriate code fragment based on the value of SHIFT_DATA. If the value is "EXPLICIT", the XVCL processor builds into the component CircularShift the necessary code to store circular shifts explicitly (as a set of strings). Otherwise, the processor builds into the component CircularShift the necessary code to store circular shifts implicitly (as pairs of index and offset). The second <select> command decides whether to include the Pairs x-frame that deals with assembling a pair of index and offset.

Accommodating the implementation-level variants

We may want to put components of different KWIC systems into different Java packages. We define a meta-variable "PACKAGE" to represent the package name, with default value of "kwic". In Figure 4, we use an XVCL command <value-of expr="?@PACKAGE?"/> to indicate a place holder at which the actual value of PACKAGE is filled during x-frame adaptation.

Unexpected changes in KWIC requirements in the future require us to add new attributes/methods to classes in KWIC components. Say, we want to add a "Health Indicator" function, which checks the number of total circular shifts in the memory and the current memory usage. If the system's free memory is low, this function will issue a warning or error message. To accommodate this kind of changes into the CircularShift component, we need new attributes and methods. We introduce two breakpoints <break name = "CIRCULARSHIFT_NEW_ATTRIBUTES"/> and <break name = "CIRCULARSHIFT_NEW_METHODS"/>. As we may also need more Java packages, we introduce a breakpoint <break name=" CIRCULARSHIFT_NEW_IMPORTS"/> to accommodate the NEW_IMPORT variant. In addition, we introduce a breakpoint NEW_PARAMETER to indicate a slot at which more XVCL parameter can be inserted. Specific code can be inserted into breakpoints by <insert> commands defined in the higher-level x-frames (e.g., in SPC) during x-frame adaptation.

7. Customizing the KWIC x-framework

We specify required configuration of variants for a KWIC system in a specification x-frame (SPC). Figure 7 shows an SPC.

```
<x-frame name="KWIC_SPC">
<set var="PACKAGE" value="kwic"/>
<set var="SHIFT_PROCESSING" value="ALLLINES"/>
<set var="SHIFTED_DATA" value="IMPLICIT"/>
<set-multi var="NOISE_WORDS" value="a,an,the"/>
<set var="INPUT_METHOD" value="FILE"/>
<set var="OUTPUT_METHOD" value="CONSOLE"/>

<adapt x-frame = "x_CircularShift"
outfile="CircularShift.java">
  <insert break="CIRCULARSHIFT_NEW_METHODS">
       public void HealthIndicator() {
           System.out.println("Total circualr
shifts in memory: " + getShiftedLines().size();
<set var="MEMORY_SAFETY_LEVEL1" value="409600"/>
<adapt x-frame="MemoryIndicator.xvcl"/>
       }
  </insert>
</adapt>

<adapt x-frame="x_Input" outfile="Input.java" />

<adapt x-frame="x_LineStorage"
outfile="LineStorage.java" />

<adapt x-frame = "x_AlphabeticShift"
outfile="AlphabeticShift.java" />

<adapt x-frame="x_Output" outfile="Output.java"/>

<adapt x-frame = "x_MasterControl"
outfile="KWIC.java">
    <insert break="KWIC_CIRCULARSHIFT_POST">
        circularShift.HealthIndicator();
    </insert>
</adapt>
</x-frame>
```

Figure 7. Specification of variants in SPC

We define XVCL meta-variables related to the variants in the <set> commands. Each meta-variable is given a specific value that meets the requirements of a particular KWIC system. These values will override the default values given in the lower-level x-frames.

In the SPC of Figure 7, we adapt the x_CircularShift x-frame by using an <insert> command. The <insert> command inserts a "Health Indicator" function mentioned in last section into the CIRCULARSHIFT_NEW_METHODS breakpoint inside the x_CircularShift x-frame. The MemoryIndicator x-frame in the <insert> section encapsulates the code about checking system memory "health" status. We adapt the MemoryIndicator x-frame (adjust the memory safety level) and then compose it into the x_CircularShift x-frame. The composed x-frame is capable of generating a CircularShift component that has the memory "health" checking functionality.

Figure 8 shows the process of constructing concrete KWIC components from x-frames. The XVCL processor customizes the x-framework according to SPC

instructions, and produces KWIC components that meet required variants. During customization, the XVCL processor traverses the x-framework, adapts visited x-frames and emits Java code for KWIC components accordingly. Figure 9 shows the CircularShift component constructed based on the SPC of Figure 7.

The SPC such as the one given in Figure 7 only contains configuration knowledge for one specific system. To develop other KWIC systems that meet different requirements, we can design different SPCs that specify values for anticipated variants and include the implementation for additional changes. In this way, we develop specific KWIC systems, members of KWIC product line.

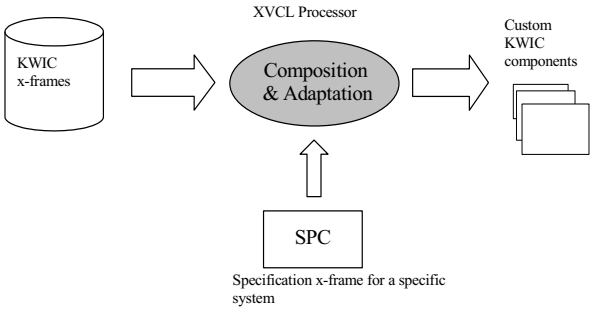

Figure 8. Customizing KWIC x-frames

```
package kwic;                    // Java Package
import java.util.*;

public class CircularShift {
   // Storage containing lines shifted
    Vector circularShifts;

    public CircularShift() {
        circularShifts = new Vector();
    }
    public Vector getShiftedLines() {
       return circularShifts;
    }
    public void circularShiftLine(String
currentLine, int LineNumber) {

     StringTokenizer parser = new
StringTokenizer(currentLine);
     int lastPosition = 0;
     while(parser.hasMoreTokens()) {
          String token = parser.nextToken();
   // Handle noise words.
   // If noise words, skip processing
          if (token.equals("a"))
              continue;
          if (token.equals("an"))
              continue;
          if (token.equals("the"))
              continue;

      lastPosition = currentLine.indexOf(token);
  //Store Circular shifts as indexes and offsets
      circularShifts.addElement(new
Pairs(lastPosition, LineNumber));
        }
        return;
    }
   … //More code about Health Indicator and Pairs
}
```

Figure 9. The CircularShift component constructed based on the SPC of Figure 7

8. Summary of experiences

The KWIC example is small. We addressed only 10 variants (and six of them have two anticipated values). Still, from the KWIC x-framework we can generate at least 2^6 different KWIC systems, given suitable SPCs. Although the time and effort we spend on constructing the x-framework is about 2-3 times of what we would spend on constructing a single system, these efforts pay off when the x-framework becomes organizational assets and is reused for developing new systems in product line.

In other projects, we have experimented with XVCL on larger scale. With our industrial partner SES Systems Pte Ltd, we have developed a Computer Aided Dispatch (CAD) product line. CAD systems are command and control systems that are used by police, fire & rescue and health service. CAD product line architecture contains 58 x-frames. We addressed 24 variants that differentiate CAD systems.

Rather than including required variants at the construction-time, we could implement variants into the component code, and provide a runtime mechanism (such as design patterns or late binding) to select required variants. The choice between construction-time and runtime techniques for handling variants is an important decision in supporting product lines, having profound impact on system properties such as complexity, reliability, performance, ease of maintenance, etc.

XVCL technique complements other contemporary software development techniques such as OO or component-based development. XVCL provides generic solutions at program construction time. Using XVCL, we avoid uncontrolled growth of components in size and number, and improve system runtime performance.

In the KWIC x-framework, x-frames contain program building blocks that separate different concerns. For example, the x_CircualrShift x-frame encapsulates the concerns of circularly shifting lines, and the sorting x-frames encapsulate the concern of sorting data. These x-frames are inter-connected via the composition and adaptation relationship, forming a layered x-framework.

XVCL shows its strength when addressing variants that have global impact on a system. The impact of such variants (such as the SHIFT_PROCESSING variant) cannot be localized into a single component identified through conventional modularization approaches such as Object-Oriented analysis and design. We accommodate these variants into x-frames by instrumenting components

with XVCL commands, and by designing lower-level x-frames that contain the impact of variants. Although certain variants may have impact on many x-frames, the x-frame processor automates the composition and adaptation process so that the transition from the x-frames to concrete components is transparent to programmers, releasing programmers from handling variants manually.

A limitation of XVCL is that, being abstract, x-frames could be difficult to understand. The verbose XML syntax also has negative impact on understanding x-frames. These problems can be mitigated by documenting the design rationale, by carefully designing x-frameworks according to XVCL design rules, and by re-factoring the design as an x-framework evolves. We are also designing XVCL Workbench to help users with XVCL-based development. The XVCL Workbench includes tools such as an x-frame editor that hides XML syntax and displays graphical views of x-frames, and a static analysis tool that helps us understand an x-framework.

9. Conclusions

We developed XVCL (XML-based Variant Configuration Language), a method and tool for product lines, to facilitate handling variants in reusable software assets. XVCL is based on "composition with adaptations" mechanism. Using XVCL, we design generic, adaptable meta-components, called x-frames. A hierarchy of inter-related x-frames forms an x-framework from which we produce software products that meet required variants. Despite its simplicity, XVCL can effectively manage a wide range of product lines variants from a compact base of meta-components, structured for effective reuse. We applied XVCL in two medium-size product line projects and a number of smaller case studies [8][14][16]. In the experiment described in this paper, we applied XVCL to handle functional, design and implementation variants in the KWIC product line. We hope this paper will encourage others to experiment with XVCL concepts and further explore the potentials of the "composition with adaptation" techniques in software engineering.

References

[1] Bassett, P. 1997. *Framing software reuse - lessons from real world*, Yourdon Press, Prentice Hall.

[2] Czarnecki, K. and Eisenecker, U., 2000. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley.

[3] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[4] Garlan, D., Kaiser, G.E. and Notkin, D., 1992. Using Tool Abstraction to Compose Systems, *IEEE Computer*, Vol. 25, June 1992, pp. 30-38.

[5] Kang, K et al. "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, CMU, Pittsburgh, Nov. 1990

[6] Kiczales, G. et al., 1997. Aspect-Oriented Programming, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Finland.

[7] Jarzabek, S. and Shubiao, L. "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," *Proc. ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, September 2003, Helsinki, pp. 237-246

[8] Jarzabek, S. and Zhang, H. "XML-based Method and Tool for Handling Variant Requirements in Domain Models", *Proc. of 5th International Symposium on Requirements Engineering*, RE'01, IEEE Press, August 2001, Toronto, Canada.

[9] Johnson, R. and Foote, B., 1988. Designing Reusable Classes, *Journal of Component-Oriented Programming*, June 1988, Vol.1, No.2, pp. 22-35.

[10] Parnas, D., 1972. On the Criteria To Be Used in Decomposing Software into Modules, *Communications of the ACM,* Vol. 15, No. 12, December, 1972, pp.1053-1058.

[11] Parnas, D., 1976. On the Design and Development of Product lines, *IEEE Trans. on Software Eng*., vol. 2, 16, pp. 1-9.

[12] Shaw, M. and Garlan, D., 1996. *Software architecture: perspectives on an emerging discipline*, Prentice-Hall.

[13] Smaragdakis, Y. and Batory, D., 2000. "Application generators", in *Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering*, J. Webster (ed.), John Wiley and Sons.

[14] Soe, M.S., Zhang, H. and Jarzabek, S. "XVCL: A Tutorial", *Proc. of 14th Int. Conf. on Software Engineering and Knowledge Engineering, SEKE'02,* ACM Press, July 2002, Italy.

[15] Wong, T.W., Jarzabek, S., Soe, M.S., Shen, R. and Zhang, H., "XML Implementation of Frame Processor," *Symposium on Software Reusability, SSR'01,* Toronto, Canada, May 2001.

[16] Zhang, H. and Jarzabek, S., "An XVCL-based Approach to Software Product Line Development", *Proc. of 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03)*, San Francisco, USA, 1 - 3 July, 2003.

Appendix – Summary of major XVCL commands

The table below summarizes major XVCL commands. The reader can find full specification of XVCL at our website fxvcl.sourceforge.net.

Table 1. Summary of major XVCL commands

Syntax	Command Definition
<x-frame name= "*name*" **>** *x-frame body* **</x-frame>**	The <x-frame> command denotes the start and end of an x-frame. An *x-frame body* contains textual contents (e.g., program code) usually instrumented with XVCL commands to make them adaptable.
<**adapt x-frame**= "*name*"> </**adapt**>	The <adapt> command instructs the processor to: • adapt the named x-frame and its descendants, • emit/assemble the customized content into the output, • resume processing of the current x-frame after processing the named x-frame and its descendants.
<**break name** = "*break-name*"> *break-body* </**break**>	The <break> command marks a breakpoint (slot) at which changes can be made via <insert>, <insert-before> and <insert-after> commands. The *break-body* defines the default code, if any, that may be replaced by <insert> or extended by <insert-before> and <insert-after> commands.
<**insert break** = "*break-name*"> *insert-body* **</insert>** <**insert-before break** = "*break-name*"> *insert-body* </**insert-before** > <**insert-after break** ="*break-name*"> *insert-body* </**insert-after** >	The <insert> command replaces the breakpoint "*break-name*" in the adapted x-frame with the *insert-body*. The <insert-before> command inserts the *insert-body* **before** the breakpoint "*break-name*" in the adapted x-frame. The <insert-after> command inserts the *insert-body* **after** the breakpoint "*break-name*" in the adapted x-frame. The *insert-body* may contain a mixture of textual content and XVCL commands.
<**set var** = "*var-name*" **value** = "*value*" **/>**	The <set> command assigns a "*value*" defined in the "value" attribute to a single-value variable "*var-name*" defined in the "var" attribute.
<**set-multi var**="*var-name*" **value**="*value1, value2, ...*" />	The <set-multi> command assigns multiple values (*value1, value2*,…) defined in the "value" attribute to a multi-value variable "*var-name"* defined in the "var" attribute.
<**value-of expr** = "*expression*" />	The value of the "*expression*" is evaluated and the result replaces the <value-of> command. A name expression starts with "?@" and ends with a "?".
<**select option** = "*var-name*"> <**option value** = "*value*"> (0 or more) *option-body* </**option**> <**otherwise**> (optional) *option-body* </**otherwise**> **</select>**	The <select> command selects from a set of options based on the value of variable "*var-name*": <**option**> is processed, if value of "*var-name*" matches <option>'s "*value*", <**otherwise**> is processed, if none of the <option>'s "*value*" is matched. The *option-body* may contain a mixture of textual content and XVCL commands.
<while using-items-in= "*multi-var*"**>** *while-body* **</while>**	The <while> command iterates over the *while-body* using the values of multi-value variable "*multi-var*" defined in the "using-items-in" attribute. The i'th iteration uses i'th value of the "*multi-var*". Inside *while-body*, *multi-var* with the i'th value can be used as a single-value variable. The *while-body* may contain a mixture of textual content and XVCL commands.