XVCL: A Tutorial

Soe Myat Swe, Hongyu Zhang and Stan Jarzabek

Department of Computer Science, School of Computing
National University of Singapore
Singapore 117543
{soemyats, zhanghy, stan}@comp.nus.edu.sg

ABSTRACT

XVCL (XML-based Variant Configuration Language) is a general-purpose mark-up language for configuring variants in programs and other types of documents. We can apply XVCL to configure variants in a variety of software assets such as software architecture, program code, test cases, technical and user-level program documentation or requirement specifications. The principles of the XVCL have been thoroughly tested in practice. XVCL is based on the same concepts as the frame technology [1]. Frame technology has been extensively applied in industry to manage variants and evolve multi-million-line, COBOL-based, information systems. An independent analysis 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, 2]. At the same time, we found that the principles of XVCL are not easy to communicate. In this paper, we describe a subset of XVCL. We trust this subset of XVCL is easy to understand and still effectively communicates essential XVCL concepts. To illustrate the XVCL method, we further describe an XVCL solution to handling variants in a Notepad system.

Categories and Subject Descriptors
D.2.13 [**Reusable Software**]: Domain Engineering

General Terms
Design, Experimentation, Languages

Keywords
XVCL, Frame Technology, Product Line

1 INTRODUCTION

XVCL (XML-based Variant Configuration Language) is a general-purpose mark-up language for configuring variants in programs and other types of documents. We can apply XVCL to configure variants in a variety of software assets such as software architecture, program code, test cases, technical and user-level program documentation or requirement specifications. In fact, XVCL can be used for managing variants in any domain that can be represented as a collection of textual documents.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy, otherwise, or republish, to post on servers or to redistribte to lists, requires prior specific permission and/or a fee.
SEKE '02, July 15-19, 2002, Ischia, Italy.
Copyright 2002 ACM 1-58113-556-4/02/0700...$5.00.

Variants arise naturally in software reuse and evolution, in particular, if you deal with software product lines that encompass a family of similar systems. This is exactly the context in which we have developed and applied XVCL. Suppose you have a software product that you want to run on different platforms or hardware devices. Or you deliver versions of the same product to a number of customers and these product versions differ in some functional or non-functional requirements. XVCL allows you to configure product variants from the common core of generic, adaptable and reusable software assets. With XVCL, you can reduce the cost of developing and evolving product lines.

However, XVCL is more than a language for configuring variants. It is accompanied by a methodology and supported by a tool – an XVCL processor. The XVCL methodology tells you how to discover the variant-structure of the solution for your application domain and for the types of variants you want to address. The XVCL processor automates what are often the most error-prone parts of program construction, allowing you to entirely focus on the essential novelty of your problems, work requiring your creativity.

Whether you develop software, manage data, design production processes, or carry out information-intensive research, themes recur with variations. Programmers, for example, commonly create new programs by modifying existing copies. A manufacturer produces multiple models of a product line, each specified by a bill-of-material that differs marginally from the other bills. A business captures customer and supplier data in forms that resemble each other. The problem is that the details that make each program, bill-of-materials or data record unique get lost, hidden within large amounts of otherwise redundant information. XVCL allows you to understand the structure of an information domain in terms of its similarities and differences by providing a means of configuring (i.e., structuring) the variations so that *all* instances can be automatically (re)constructed from their underlying themes. XVCL is independent of any specific programming language or application domain and can be useful in both software and non-software domains.

XVCL is a scripting language that allows you to specify how to systematically and reliably modify programs at variation points in order to accommodate specific variants into programs. You use XVCL commands to mark variation points in your program. To facilitate effective reuse, you split your program into generic, adaptable fragments, called x-frames. Each x-frame is instrumented with XVCL commands to permit automatic customization and evolution. You organize x-frames into a hierarchy that forms an adaptable architecture for your product line.

The XVCL processor traverses an x-frame hierarchy and performs adaptation by executing XVCL commands embedded in x-frames. During processing, each x-frame, in effect, *adapts* the x-frames of its (sub) hierarchy to produce a specific system, a member of the product line. XVCL processor assembles customized x-frames into a program that meets specific variants.

The principles of the XVCL have been thoroughly tested in practice. XVCL is based on the same concepts as the frame technology [1]. Frame technology has been extensively applied in industry to manage variants and evolve multi-million-line, COBOL-based, information systems. While designing a frame architecture is not trivial, subsequent complexity reductions and productivity gains are substantial. An independent analysis 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, 2]. These gains are due to the flexibility of the resulting architectures and their evolvability over time. The excellent record of frame technology in large-scale software applications was the main reason which led us to implementing XVCL [8].

XVCL is extensible and, of course, free of COBOL heritage. We applied XVCL to handle variants and evolve component-based systems written in Java [6, 3] and to manage variants in UML software models documenting product lines in Facility Reservation and Computer Aided Dispatch system domains [7]. Skillfully structured XVCL solutions collapse the size of the problem so that typically you need to focus on only the 5%-15% of the program solution that is unique; the other 85%-95% is constructed automatically.

XVCL facilitates change. For example, modifying programs is tedious and notoriously error-prone. XVCL not only greatly speeds up the modification process, but also performs all changes in a much more reliable manner – it never gets sloppy or forgets to make an intended change. XVCL helps you reuse information and control its evolution over time.

Despite effectiveness of XVCL in solving practical problems, the principles of XVCL are not easy to communicate. In this paper, we describe a subset of XVCL, which is easy to understand and still effectively communicates essential XVCL concepts. To illustrate the XVCL method, we further describe an XVCL solution to handling variants in Notepad systems. Finally, we compare XVCL with other methods for handling variants in product lines.

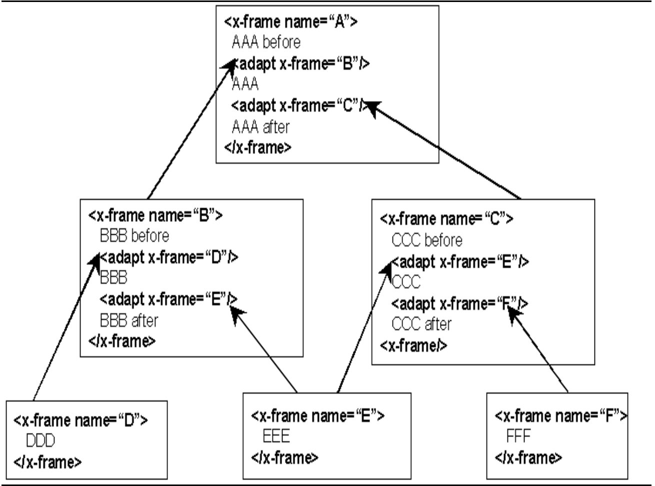
2 SALIENT FEATURES OF XVCL

2.1 XVCL Fundamentals

In XVCL, generic, adaptable components are called x-frames. An *x-framework* is a collection of inter-related x-frames that we reuse in the construction of programs or product lines. The x-frames in an x-framework may be linked together by XVCL <adapt> commands (explained in section 2.2), provided no <adapt> chains loop back on themselves. X-frameworks are more general than trees, but as in a tree, every x-subframework will have a unique *x-frame* as its *root.*

An x-frame is an XML file with program code instrumented with XVCL commands for ease of customization. X-frames in our examples will contain Java code. XVCL commands must follow the rules of the XVCL language to be processed by the XVCL processor. XVCL is based on XML[1] so the usual XML rules apply to XVCL. An x-frame is valid if it conforms to the rules specified in its corresponding Document Type Definition (DTD). The DTD defines the XVCL grammar that specify valid XVCL commands, their corresponding attributes and nesting structures.

XVCL is supported by a processor. Before processing x-frames, the processor checks if x-frames conform to the grammar definitions in the DTD. If they do, the processor traverses x-framework, interprets XVCL commands embedded in visited x-frames and assembles the output (e.g., a custom program) into one or more files. In our example, the output is emitted to a single file. The processor's traversal order is dictated by <adapt> commands embedded in x-frames. This customization process of the x-framework is directed by instructions contained in a specification x-frame, called an SPC for short. (Each SPC specifies a different customization of an x-framework.)



```
<x-frame name="A">
 AAA before
 <adapt x-frame="B"/>
 AAA
 <adapt x-frame="C"/>
 AAA after
</x-frame>
```

```
<x-frame name="B">
 BBB before
 <adapt x-frame="D"/>
 BBB
 <adapt x-frame="E"/>
 BBB after
</x-frame>
```

```
<x-frame name="C">
 CCC before
 <adapt x-frame="E"/>
 CCC
 <adapt x-frame="F"/>
 CCC after
<x-frame/>
```

```
<x-frame name="D">
 DDD
</x-frame>
```

```
<x-frame name="E">
 EEE
</x-frame>
```

```
<x-frame name="F">
 FFF
</x-frame>
```

Figure 1. An example of an x-framework and SPC

We shall now illustrate the customization process. In the example of Figure 1, x-frames A (an SPC), B, C, D, E and F form an x-framework rooted at x-frame A, in which:

- x-frame A adapts x-frames B and C,
- x-frame B adapts D and E,
- x-frame C adapts E and F.

X-frames B and C are roots of their respective x-subframeworks.

The <adapt> command tells the processor to customize the specified x-frame and assemble the customized result into the output. Figure 2 shows the traversal path of the XVCL processor for the example shown in Figure 1. The custom result is shown on the right hand side of the Figure 2.

[1] For more information on XML please refer to "www.w3c.org/TR/REC-xml"

What we did not show in the above example, is that <adapt> command may also specify customization commands. In such a case, part of the output will be a customized version of the <adapt>ed x-frame. We shall discuss this in more detail below.

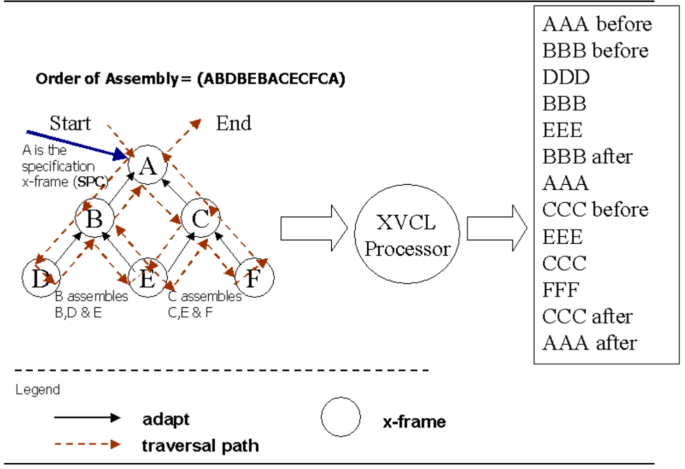


Figure 2. Example of x-frame processing

2.2 Description of Essential XVCL Commands

We shall now describe a subset of XVCL. The reader can find full specification of XVCL at our Web site: www.comp.nus.edu.sg/labs/software/xvcl.html

Syntax	**Attribute Definition**	**Command Definition**
<x-frame name= ”*name*” **>** *x-frame body*: mixture of code and XVCL commands **</x-frame>**	**name:** is the name of the x-frame being defined.	The <x-frame> command denotes the start and end of the x-frame body. The x-frame body contains textual contents (e.g., program code), instrumented with XVCL commands for ease of adaptation.
<**adapt x-frame**=”*name*”> *adapt-body* : mixture of <insert>, <insert-before>, <insert-after> commands </**adapt**> or: <**adapt x-frame**=”*name*”/>	**x-frame:** defines the name of x-frame to be adapted.	The <adapt> command instructs the processor to: • adapt the x-subframework rooted in the named x-frame by inserting text contained in <insert> commands at specific breakpoints of <adapt>ed x-frames, • emit/assemble the customized content of the adapted x-subframework into the output, • resume processing of the current x-frame after processing the x-subframework rooted in the named x-frame. The *adapt-body* may contain a mixture of <insert>, <insert-before> and <insert-after> commands.
<**break name** =”*break-name*”> *break-body* </**break**> or: <**break name** =”*break-name*”/>	**name**: defines the name of the breakpoint in an x-frame.	The <break> command marks a breakpoint at which changes can be made by ancestor x-frames 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** >	**break**: defines the name of the breakpoint.	The <insert> command replaces the breakpoints “*break-name*” in the adapted x-subframework with the *insert-body*. The <insert-before> command inserts the *insert-body* **before** the breakpoints “*break-name*” in the adapted x-subframework. The <insert-after> command inserts the *insert-body* **after** the breakpoints “*break-name*” in the adapted x-subframework. The *insert-body* may contain a mixture of textual content and XVCL commands.
<**set var** = ”*var-name*” **value** = ”*value*” />	**var**: defines the name of the single-value variable. **value**: defines the value to be assigned.	The <set> command assigns a “*value*” defined in “value” attribute to single-value variable “*var-name*” defined in “var” attribute.
<**set-multi var**=”*var-name*” **value**=”*value1, value2, …*” />	**var**: defines the name of the multi-value variable. **value**: defines a list of values to be assigned to the variable.	The <set-multi> command assigns multiple values (*value1, value2,…*) defined in “value” attribute to a multi-value variable *“var-name”* defined in “var” attribute.
<**value-of expr** = ”*expression*” />	**expr**: defines an expression to be	The value of the “*expression*” is evaluated and the result replaces the

	evaluated.	<value-of> command.
<**select option** = ”*var-name*”> *select-body*: may contain options listed below </**select**> *select-body*: <**option-undefined**> (optional) *option-body* </**option-undefined**> <**option value** = ”*value*”> (0 or more) *option-body* </**option**> <**otherwise**> (optional) *option-body* </**otherwise**>	**option**: The “option” attribute in <select> command defines the variable whose value will be matched in <option> commands. **value**: The “value” attribute in <option> command defines the value to be matched.	In this command, we select from a set of options based on variable “*var-name*” as follows: • <**option-undefined**> is processed, if the variable “*var-name*” is undefined, • <**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>**	**using-items-in**: defines the multi-value variable “*multi-var*” to be used inside while.	The <while> command iterates over the *while-body* using the values of multi-value variable “*multi-var*” defined in “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 single-value variable. The *while-body* may contain a mixture of textual content and XVCL commands.
<!-- comment -->		Text enclosed between <!-- --> is considered a comment. Comments may spread over multiple lines.

Table 1. Essential XVCL commands

2.3 XVCL Expressions and Variable Scoping Rules

Generic names increase flexibility and adaptability of programs and play an important role in building generic, reusable programs. XVCL variables and expressions provide powerful means for creating generic names and controlling the x-framework customization process.

An XVCL expression may involve direct and indirect references to XVCL variables, name expressions and concatenation of name expression values with strings of characters.

2.3.1 References to variables

A direct reference to variable C is written as: @C.

Each extra symbol ‘@’ in the front of a variable name indicates a level of indirection. So:

@@C means value-of (value-of (C))

@@@C means value-of (value-of (value-of (C))), etc.

Variable references are replaced by their respective values. Here, we should mention, that XVCL processor stores all the existing variables in the Symbol Table along with their current values, as assigned to variables in <set> commands. A reference to a non-existing variable is considered an error.

For example, referring to Symbol Table (Table 2), the value of @@C is BU and the value of @@@C is V.

2.3.2 XVCL expressions

We shall introduce name expressions first and then explain expressions in their full form. A simple name expression may contain just a variable reference, such as: ?@C? or ?@@C?. (A name expression is always enclosed between question mark symbols ‘?’.)

More complex (but more useful) name expressions can be written as: ?@A@B@C?. The value of such a name expression is computed from right to left as follows:

value-of (“A” | value-of (“B” | value-of (C))), where symbol ‘|’ means string concatenation.

After each evaluation step, the intermediate value computed is concatenated with the character string on the left to form a new variable name that is looked up in the Symbol Table. Evaluation of a name expression continues until the whole name expression is evaluated. A name expression may contain a mixture of indirect references and direct references to variables and any number of concatenations.

In a valid name expression, the rightmost string and also strings created at each intermediate evaluation step must represent variables that exist in the Symbol Table. All such variables must have been defined in <set> commands before a given name expression is evaluated.

The following example illustrates evaluation of name expressions. Suppose when we evaluate name expression: ?@A@B@C?, the Symbol Table contains variables shown in Table 2.

Variable Name	*Value*
A	X
X	Y
Y	Z
C	U
U	BU
BU	V
AV	W

Table 2. The Symbol Table with XVCL variables

The evaluation of a name expression ?@A@B@C? is done as follows:

1. get the value of variable C
    - the intermediate result is U
2. concatenate B and U and get the value of variable BU
    - the intermediate result is V
3. concatenate A and V and get the value of variable AV
    - the final result is W.

XVCL expressions in their full form may contain any number of name expressions intermixed with character strings. To evaluate an expression, we evaluate all the name expressions and concatenate resulting values with character strings. The result replaces the corresponding expression.

As an example, we shall evaluate an expression: ?@A@B@C?P?@X? assuming variable values indicated in Table 2:

1. evaluate name expression ?@A@B@C?
    - the result is W
2. replace ?@A@B@C? with W
    - partially evaluated string becomes WP?@X?
3. evaluate name expression ?@X?
    - the result is Y
4. replace ?@X? with Y
    - the final result is WPY

2.3.3 Variable scoping rules

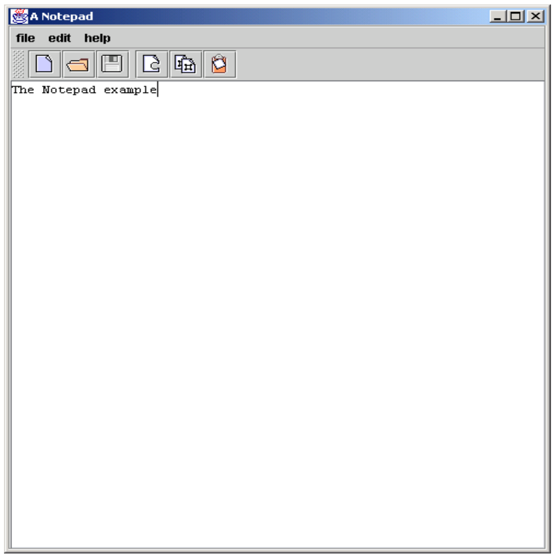
Variable scoping rules are the same for both single-value and multi-value variables. The <set> command(s) in the ancestor x-frame takes precedence over <set> commands in its descendent x-frames. That is, once an x-frame X sets the value of variable v, <set> commands that define the same variable v in descendent x-frames (if any) visited by the processor will not take effect. However, the subsequent <set> commands in x-frame X can reset the value of variable v.

Variables become undefined as soon as the processing level rises above the x-frame that effectively set variable values. (Note: variables that are set within <insert> commands become undefined when the processing level rises above the x-frame containing the <break>.) This makes it possible for other x-frames to set and use the same variables and prevents interference among variables used in two different x-subframeworks in the x-framework.

The above scoping rules are important for reuse. So that they can be reused in many systems, lower level x-frames are usually generic, meaning that they contain variables, <break>s, and <select>s. Such x-frames use <set> commands to define default values of variables to produce a default output text. When an ancestor x-frame needs to adapt such an x-frame to its context, it can use <set> commands to override one or more of the defaults, thereby customizing the output text.

3 NOTEPAD EXAMPLE

This example uses the development of a generic text editor (Notepad) to illustrate XVCL concepts and the usage of XVCL commands. A simple Notepad is presented in Figure 3.



```
The Notepad example
```

Figure 3. A simple Notepad

This Notepad is a typical Java Window, with a menu-bar, a toolbar and an editing panel inside. It supports basic text editor functionalities such as opening and editing a file. However, our objective is not to develop just one-of-a-kind Notepad, but a generic and flexible Notepad system so that:

1) Other similar systems (members of Notepad product line) can be easily developed from it.
2) It can cater for the changes arising from system maintenance and evolution.

The above two situations result in many variant requirements, such as:

- Notepad may have more menus (and menu items) or toolbar buttons to support more functionality.
- The title (name) of the Notepad may change.
- Different appearance ("look and feel") of the Notepad (e.g., Microsoft Window style, Motif style, etc.).
- The background color of the Notepad may vary.

How does one design the Notepad so that it can be easily customized/changed to meet the variant requirements? In this section, we shall illustrate a solution to this problem in XVCL. Due to space constraints, we only show how we handle the toolbar, title, and background variant requirements. The complete description of the Notepad example can be found at our website at www.comp.nus.edu.sg/labs/software/xvcl.html.

3.1 Notepad x-frame

Figure 4 shows an x-frame for Notepad. It contains the default code that is common to all Notepads, as well as XVCL commands that indicate the variation points.

```
<x-frame name="Notepad">

<!-- default settings for variables in this x-
frame: -->

<set var="TITLE" value="Notepad"/>
<set var="BGCOLOR" value="gray"/>

<break name="NOTEPAD_NEWPARAMETERS"/>

import java.awt.*;
import java.awt.event.*;
import java.util.*;

import javax.swing.text.*;
import javax.swing.event.*;
import javax.swing.*;

<break name="NOTEPAD_NEWIMPORTS"/>

class Notepad extends JPanel {

    Notepad() {
      super();
      …
    }

    public static void main(String[] args) {
       JFrame frame = new JFrame();
       frame.setTitle("<value-of
expr="?@TITLE?"/>");
      frame.setBackground(Color.<value-of
expr="?@BGCOLOR?"/>);
       …
       frame.show();
    }
    <adapt x-frame="Editor.XVCL"/>
    <adapt x-frame="Menubar.XVCL"/>
    <adapt x-frame="Toolbar.XVCL"/>
       …
    <break name="NOTEPAD_NEWMETHODS"/>
}

</x-frame>
```

Figure 4. Notepad.XVCL

A typical Notepad is composed of many components such as an editor, a menubar, a toolbar, etc. We design an x-frame for each of these components. Each x-frame can be separately reused and maintained. We include these x-frames into the Notepad x-frame by using the <adapt> command.

Different Notepads may have different names. We use XVCL variable TITLE to represent the name of the Notepad. The first <set> command assigns a default value "Notepad" to variable TITLE. Command `<value-of expr="?@TITLE?"/>` indicates a place holder where the default value of the variable TITLE can be substituted at program construction time. Similarly, we use XVCL variable BGCOLOR to handle variations in Notepad's background color.

The `<break name="NOTEPAD_NEWPARAMETERS"/>` command indicates a breakpoint at which the declarations of additional XVCL variables may be inserted. Similarly, the `<break name="NOTEPAD_NEWMETHODS"/>` command indicates a breakpoint at which possible new methods could be inserted.

3.2 Toolbar x-frame

Looking at Notepad's toolbar carefully you may discover that there are many commonalities among various tool buttons. Basically, each tool bar has a name, an icon (stored as a .gif file) and an associated action. Code for creating one tool button is very similar to the code for creating other tool buttons. This motivates us to design a generic solution for creating all kinds of tool buttons and toolbars. Figure 5 shows an x-frame Toolbar.XVCL from which we can generate Java code for creating toolbars.

Comments on x-frame Toolbar.XVCL of Figure 5: Each button in the tool bar has an icon, a tip and an associated action. For each tool button, we must create its icon, link the button to a tool tip and to an action. Code for creating tool buttons is generated in the <while> loop. We use a multi-value variable ToolbarBtns to represent the buttons in the toolbar. Depending on the value of the variable ToolbarBtns (referred to in expression ?@ToolbarBtns?), the corresponding code for creating tool buttons is generated (If the value is "-", the code for creating a separator is generated). The name expression ?@Gif@ToolbarBtns? represents a generic tool button icon. Icons are defined in corresponding .gif files. Based on the value of the variable ToolbarBtns (e.g., New, Open, Save, Exit), a file containing the proper tool button icon is selected. The name expression ?@Tip@ToolbarBtns? represents a generic tool button tip and is used to assign tips to tool buttons. Expression "?@Action@ToolbarBtns?() " represents a generic tool button action. The value of name expression ?@Action@ToolbarBtns? is evaluated and concatenated with brackets "(" and ")" to create the name of a method that will be invoked when a particular button is clicked.

```
<x-frame name="Toolbar">

<!-- default settings for multi-value variable ToolbarBtns: -->
<set-multivar="ToolbarBtns" value="New,Open,Save, -, Exit"/>

<break name="TOOLBAR_NEWPARAMETERS"/>

<!-- Create a Java toolbar  -->
    private Component createToolbar() {
           JToolBar toolbar = new JToolBar();
           JButton button;
<while using-items-in="?@ToolbarBtns?">
    <select option="?@ToolbarBtns?">
```

```
       <option value="-">
          toolbar.add(Box.createHorizontalStrut(5));//Creating a separator between two buttons
       </option>
       <otherwise>
              // Creating a button icon
           button = new JButton(new ImageIcon("<value-of expr="?@Gif@ToolbarBtns?"/> "));
              //linking the   button to a tool tip
           button.setToolTipText("<value-of expr="?@Tip@ToolbarBtns?"/>");
           button.addActionListener(new java.awt.event.ActionListener() {
                         public void actionPerformed(ActionEvent e) {
                             <value-of expr="?@Action@ToolbarBtns?()"/>;
                         }
                     });    //linking the button to an action
           toolbar.add(button);
        </otherwise>
    </select>
</while>
           toolbar.add(Box.createHorizontalGlue());
           return toolbar;
    }

<break name="TOOLBAR_ACTIONS">
    <while using-items-in="?@ToolbarBtns?">
            <select option="?@ToolbarBtns?">
                 <option value="-">
                  </option>
                  <otherwise>
                     <adapt x-frame="?@Action@ToolbarBtns?.XVCL"/>
                  </otherwise>
             </select>
    </while>
</break>

</x-frame>
```

Figure 5. Toolbar.XVCL

The <while> loop iterates over tool buttons and assigns icons, tips and actions to them, one by one.

We use the command <adapt x-frame="?@Action@ToolbarBtns?.XVCL"/> to adapt x-frames containing the implementation of actions for various tool buttons. An example of x-frame for the "New File" action is shown below.

```
<x-frame name="NewFile">
private void NewFile() {

<break name="NEWFILE">
    editor.setDocument(new PlainDocument());
    revalidate();
</break>

    return;
}
</x-frame>
```

Figure 6. NewFile.XVCL

3.3 Notepad x-framework

An overview of a Notepad x-framework is shown in Figure 7. It has a tree-like hierarchical structure, where upper layer x-frames adapt lower layer x-frames.

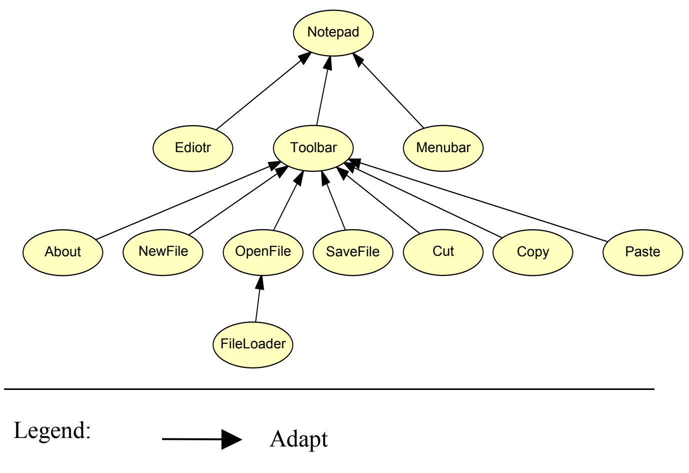


Figure 7. The Notepad x-framework

3.4 Specification x-frame (SPC)

The specification for a particular Notepad system is described in the specification x-frame (SPC). An SPC tells the XVCL processor how to customize x-frames to generate the code that meets the specific requirements. Figure 8 shows an SPC for a specific Notepad. This SPC defines the title of the Notepad as "A

Notepad", and defines the background color as "lightGray". The SPC also defines six buttons (new, open, save, cut, copy, and paste) with their associated icons (.gif files), tips and actions. These specifications describe a specific Notepad that is needed. Given the SPC, the XVCL processor traverses and customizes the Notepad x-framework to generate Java code for a specific Notepad.

```
<x-frame name="Notepad"
outfile="Notepad.java" language="java">
      …
<set var="TITLE" value="A Notepad"/>
<set var="BGCOLOR" value="lightGray"/>

<set-multivar="ToolbarBtns"
value="New,Open,Save,-,Cut,Copy,Paste"/>

<set var="GifNew"
value="resources/new.gif"/>
<set var="GifOpen"
value="resources/open.gif"/>
<set var="GifSave"
value="resources/save.gif"/>
<set var="GifCut"
value="resources/cut.gif"/>
<set var="GifCopy"
value="resources/copy.gif"/>
<set var="GifPaste"
value="resources/paste.gif"/>

<set var="TipNew" value="Create a file"/>
<set var="TipOpen" value="Open a file"/>
<set var="TipSave" value="Save to a file"/>
<set var="TipCut"  value="Move selection to
clipboard"/>
<set var="TipCopy" value="Copy selection to
clipboard"/>
<set var="TipPaste" value="Paste clipboard
to selection"/>

<set var="ActionNew"  value="NewFile"/>
<set var="ActionOpen" value="OpenFile"/>
<set var="ActionSave" value="SaveFile"/>
<set var="ActionExit" value="Exit"/>
<set var="ActionCut"  value="Cut"/>
<set var="ActionCopy" value="Copy"/>
<set var="ActionPaste" value="Paste"/>

<adapt x-frame="Notepad.xvcl"/>

</x-frame>
```

Figure 8. SPC Notepad.S

Comments on SPC Notepad.S of Figure 8: The multi-value variable ToolbarBtns defines buttons (such as New, Open, Save, etc.) that make up a specific tool bar:

> <set-multivar="ToolbarBtns" value="New,Open,Save,-,Cut,Copy,Paste"/>

Variable ToolbarBtns drives <while> loops in the Toolbar.xvcl x-frame (Figure 5).

Six commands starting with: <set var="GifNew" value="resources/new.gif"/> define .gif files containing tool button icons. After that, the reader can find similar commands for button tips and actions.

The Notepad x-framework facilitates evolution that may occur in the future. Certain types of changes can be easily accommodated into the Notepad x-framework by modifying values of suitable XVCL variables. To address more drastic unexpected changes, we may need to adapt x-frames at breakpoints (via <insert> commands in the SPC) or add new breakpoints in x-frames.

While all the changes can be kept separately from the affected x-frames (in SPCs), x-frames themselves also do evolve over time. As requirements change, we may need to re-design some of the x-frames and extend them to enable reuse of new functions.

4 COMPARISON OF XVCL AND OTHER APPROACHES TO PRODUCT LINES

4.1 Templates

Templates evolved from macros in C language. Templates (particularly in C++) facilitate reuse by making the underlying program functionality independent of data types. Templates support selection and composition of functions and classes by creating conditional control selection (e.g., IF<> and SWITCH<> templates) and repetitions (e.g., WHILE<>, DO<> and FOR<> templates). Although these mechanisms allow certain degree of flexibility in template composition, they can only cater for anticipated changes.

XVCL, on the other hand, can cater for both anticipated changes (with <select>s) and unexpected changes (with <break>s). Being an advanced form of a macro system, XVCL facilitates template-like meta-programming [4] and code generation. X-frames may be viewed as highly configurable parameterized templates or meta-components. Not only do they provide for variability of data types in classes and functions, but also for variability in any functional and non-functional system requirements. Each x-frame localizes the impact of change by providing default composition rules and configuration knowledge.

Templates support decomposition of a program along class, function and parameter boundaries. X-frames on the other hand, are language independent and support arbitrary decomposition and composition of a program (such as having generic data structures defined independently of the generic methods that use them). Thus, it is not only easy to write templates in XVCL but it also permits a stronger separation of concerns.

4.2 Software Configuration Management (SCM)

A number of studies have been carried out on the role of SCM systems in software reuse (e.g., PCL). Most of the SCM systems concentrate on managing versions of source and binary files and identifying correct versions of these files in system building.

PCL [5], a configuration language proposed by Gulla and Floch, enables instantiation of highly parameterized components by allowing users to configure component parameters and component compositions. This achieves a certain degree of component reuse. However, components' parameterization mechanism is not sufficient to achieve high levels of reuse. Any given component may need to satisfy different combinations of properties. For each legal combination of properties, a PCL system maintains a separate component version. In XVCL, on the other hand, x-frames capture component variability in a same-as-except manner rather than keeping different versions for different property combinations. X-frames also facilitate configuration of variants in any level of granularity in program source code. This prevents the number of components in the x-framework from exploding.

One of the strong points of SCM systems is their parallel and concurrent version control mechanism, very much different from the version control mechanism in XVCL. XVCL provides implicit version control, where different versions of systems are captured and produced by means of configuration and adaptation knowledge of x-frames.

4.3 Macro Systems

Macro systems are probably the oldest form of meta-programming. They have been integrated into both procedural and object-oriented languages to facilitate writing more flexible and reusable programs. XVCL is as an advanced form of macro system. XVCL variable is a counterpart of macro parameter. The <adapt> command in XVCL is similar to a macro call, while the <set> command corresponds to setting a value of a macro parameter. XVCL decouples variables from an x-frame itself – an x-frame can refer to any variable in its ancestor x-frames. Therefore unlike in macros, variables set in a higher level x-frame can be passed to lower level x-frames without involving intermediate x-frames.

In XVCL, variables are local to an x-frame, meaning that only an x-frame that first sets the value of a variable can reset its value. Once the value of a variable has been set, the variable is "read only" to its descendent x-frames. The aim of this variable scoping rule is to keep lower level x-frames as generic (meaning context free, reusable) as possible. SPC x-frame (and other higher level x-frames) can adapt any lower x-frames to produce the customized system.

In XVCL, variables become undefined as soon as the processing level rises above the x-frame that effectively set variable values. This makes it possible for other x-frames to set and use the same variables and prevents interference among variables used in two different subtrees in the x-framework.

In macros, there is no concept of default values for parameters (other than null values). In XVCL, this concept plays an important role in fostering reuse. All the variables in x-frames have default values as defined by <set> commands. Only the variables with default values that need to be changed are passed by the ancestor x-frame. The rest are used unchanged. This greatly reduces the complexity that the ancestor x-frames have to handle.

A macro cannot customize the content of other macros. This limits reusability of macros– if a macro does not fit a particular reuse context, it cannot be adapted. X-frames, on the other hand, can customize other x-frames for reuse using <insert> and <break> commands.

Once the structure of macro composition is established, it cannot be changed. We cannot easily add, remove or change macro calls in existing macros. On the other hand, XVCL allows one to modify x-frame calling structure by inserting new <adapt> commands into x-frames. The <insert> command can insert XVCL commands such as <adapt> at the <break>s in the lower-level x-frames. Those commands are executed as if they were originally declared in affected x-frames. This customization mechanism adds extra flexibility to x-frames, making them reusable in many contexts.

5 CONCLUSIONS

XVCL (XML-based Variant Configuration Language) is a general-purpose mark-up language for configuring variants in programs and other types of documents. In this paper, we described a subset of XVCL. We trust this subset of XVCL is easy to understand and still effectively communicates essential XVCL concepts. To illustrate the XVCL method, we further described an XVCL solution to handling variants in a Notepad system. The principles of the XVCL have been thoroughly tested in practice. XVCL is based on the same concepts as the frame technology [1]. Frame technology has been extensively applied in industry to manage variants and evolve multi-million-line, COBOL-based, information systems. An independent analysis 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, 2]. At the same time, we found that the principles of XVCL are not easy to communicate. The purpose of this paper is to fill this gap. In particular, the comparison with the more familiar programming mechanisms of template, configuration management, and macro, should help to clarify the ideas.

6 REFERENCES

[1] Bassett, P. *Framing Software Reuse - Lessons From Real World*, Yourdon Press, Prentice Hall, 1997.

[2] Bassett, P. "The Theory and Practice of Adaptive Components", *Proc. 2nd International Symposium on Generative and Component-Based Software Engineering, GCSE 2000*, Erfurt, Germany, October 9-12, 2000, Springer Lecture Notes in Computer Science, LCNS2177, (Eds. G. Butler and S. Jarzabek), pp. 1-14

[3] Cheong, Y.C. and Jarzabek, S. "Frame-based Method for Customizing Generic Software Architectures", *Proc. Symposium on Software Reusability, SSR'99,* Los Angeles, May 1999, pp. 103-112

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

[5] Floch, J. and Gulla, B. "Enabling reuse with a Configuration Language", *Proc. of the Fourth International Conference on Software Reuse (ICSR '96)*, Orlando, FL, 1996.

[6] Jarzabek, S. and Seviora, R. "Engineering components for ease of customization and evolution," *IEE Proceedings - Software*, Vol. 147, No. 6, December 2000, pp. 237-248, a special issue on Component-based Software Engineering.

[7] Jarzabek, S. and Zhang, H. "XML-based Method and Tool for Handling Variant Requirements in Domain Models", *Proc. of 5th IEEE International Symposium on Requirements Engineering, RE'01,* IEEE Press, August 2001, Toronto, Canada, pp. 166-173

[8] Wong, T.W., Jarzabek, S., Myat Swe, S., Shen, R. and Zhang, H.Y. "XML Implementation of Frame Processor," *Proc. ACM Symposium on Software Reusability, SSR'01,* Toronto, Canada, May 2001, pp. 164-172