ANALYSIS AND DEBUGGING OF META-PROGRAMS IN XVCL

CHEN LIANG

NATIONAL UNIVERSITY OF SINGAPORE

2004

ANALYSIS AND DEBUGGING OF META-PROGRAMS IN XVCL

CHEN LIANG

(B.Comp. (Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

First and foremost, I am deeply indebted to my supervisor, Dr. Stanislaw Jarzabek, Associate Professor, School of Computing, for teaching me software engineering knowledge and giving me the opportunity to work on this project. He has been helpful and provided the invaluable guidance throughout the project.

I would also like to express my heartfelt thanks to Mr. Ulf Pettersson and Soe Myat Swe, SES Pte Ltd, for their valuable discussions and inputs.

In addition, I would like to thank Ms. Cao Yang, Research Assistant, Software Engineering Lab, for her help and discussions.

I am also grateful to Mr. Damith C. Rajapakse and Hamid Abdul Basit, my teammates, for many lively discussions to clarify the knowledge and share ideas.

Lastly, I would like to thank my parents and my brother for their love and encouragement and understanding.

Table of Contents

List of Figures

Summary

XML-based Variant Configuration Language (XVCL for short) is a meta-programming language and technique for effective software reuse. It is based on the frame technology, expressed in XML conventions. XVCL is more than a language for configuring variants. It is accompanied by a methodology and supported by a tool – an XVCL processor.

In XVCL, we partition programs into generic, adaptable meta-components called x-frames and organize x-frames into a hierarchical structure called an x-framework. Understanding an x-framework is time consuming and error-prone. This problem impedes XVCL's usability and becomes an obstacle to effective application of the XVCL technology. This problem, however, can be alleviated by a proper tool that helps us analyze x-frameworks at runtime.

The XVCL processor works in the batch mode – to generate a custom program, it processes an x-framework without stopping. We call it the batch XVCL processor throughout our project. The batch XVCL processor plays well its role when everything goes fine during the custom program generation. But in case of errors, we need trace and understand the processing sequence, in an interactive way.

In this project, we formulate general requirements for analysis and debugging of meta-programs. Based on that, we develop a tool – an interactive XVCL processor (IXP for short) for analysis/debugging meta-programming in XVCL. IXP is a graphical tool that helps users develop, analyze, modify and debug XVCL meta-programs. IXP is developed based on the batch XVCL processor.

In this thesis, we present our findings in the area of analysis and debugging of meta-programs.

We describe unique features of IXP. These features include:

- the ability to trace the program generation process in a flexible way. We developed a specification language allowing a user to specify debugging points (a debugging point is a point at which the processing suspends allowing a user to inspect the state of processing)
- the ability to inspect and modify values of XVCL variables at the debugging points
- the ability to define undefined meta-components which allows the user to process incomplete meta-programs
- the ability to step through a meta-program in multiple ways, as required to understand the generic nature of a meta-program
- the ability to store useful trace information resulting from processing for further analysis

In this thesis, we also briefly describe the design and implementation of IXP.

Chapter 1

Introduction

1.1 Background

Developing software systems from the scratch is costly and time consuming. Software reuse methods proposed in the last decade considerably improve software quality and productivity. Effective reuse can be achieved by applying product line approach, which was initiated by Parnas back in 1970s [10].

A software product line is a family of systems that have similar characteristics and requirements [3]. Product line members share many software assets such as software components, documentations, software architecture, process models and test cases that can be reused across the members of a product line. Instead of focusing on a single system, the product line approach focuses on developing, organizing and managing reusable software assets for a group of systems that make up a particular product line.

While having much in common, members of a product line also differ in certain requirements, design specifications, platforms and other characteristics. The variability comes from many sources such as users' specific needs, mutability of the environment, system evolution and so on. Variants in a product line are the results of the variability among members of this product line.

In the product line approach, we identify both commonalty and variability, and build reusable assets across product line members. In general, variability is more difficult to handle than commonalities [5]. Variants arise naturally in software evolution. Particularly in product line situation, we must manage multiple software requirement specifications, software models, designs, codes, test cases and documentations for different products. As the volume of information and number of variants grow, it becomes increasingly difficult to manage all these software assets in a systematic way.

To handle variants in product line assets, a mark-up language called XML-based Variant Configuration Language (XVCL for short) was introduced (Software Engineering Lab, National University of Singapore, 2001). XVCL is a meta-language. Meta-programs created with XVCL are generic, incomplete and adaptable [6]. XVCL is method and tool to facilitate software reuse. With XVCL, users can integrate variants into all kinds of product line assets such as architecture, code components, documentation, even UML models.

XVCL is based on industry-proven concepts of Bassett's frames [2]. Frames have achieved substantial productivity improvements in large data processing product lines. XVCL blends with contemporary programming paradigms, offering an effective reuse mechanism complementing other design techniques. XVCL is based on a "composition with adaptation" mechanism [12].

However, XVCL is more than a language for configuring variants. It is accompanied by a methodology and supported by a tool – an XVCL processor [14]. It takes more than knowing XVCL rules to create successful XVCL solutions to problems of managing variants in users' application domain. The XVCL processor automates routine tasks for users.

1.2 Project objectives and scope

In XVCL, we partition programs into generic, adaptable meta-components called x-frames and organize x-frames into a hierarchical structure called an x-framework.

Understanding an x-framework is time consuming and error-prone. This problem impedes XVCL's usability and becomes an obstacle to effective application of the XVCL technology.

This problem, however, can be alleviated by a proper tool that helps us analyze x-frameworks at runtime.

The XVCL processor works in the batch mode – to generate a custom program, it processes an x-framework without stopping. We call it the batch XVCL processor throughout our project. The batch XVCL processor plays well its role when everything goes fine during the custom program generation. But in case of errors, we need trace and understand the processing sequence, in an interactive way.

The objective of the project is to develop the interactive XVCL processor (IXP, for short) based on the batch XVCL processor. IXP is a graphical tool that helps users develop, analyze, modify and debug meta-programs created with XVCL.

Our project comprises of three stages:

- understanding concepts of XVCL
- analyzing the batch XVCL processor and understanding its implementation
- developing the interactive XVCL processor (IXP)

In the first stage, we study basic concepts of XVCL. This involves understanding XVCL fundamentals, studying XVCL commands, performing conceptual analysis with help of examples.

In the second stage, we analyze the batch XVCL processor by reading its source code. We also develop some examples to illustrate how the batch XVCL processor works.

In the third stage, we implement IXP based on the batch XVCL processor. This includes design, implementation and thorough testing.

1.3 Contributions of the project

The main contribution of this thesis is in setting up a stage for automated analysis/debugging of static meta-programming systems. Based on general requirements for analysis/debugging of meta-programs, we developed an analysis/debugging method and tool, called IXP, for meta-programming in XVCL. While the details of analysis/debugging methods depend on a specific meta-programming technique, we believe our results are interesting and useful in general. Specific contributions include: (1) identifying types of questions programmers ask to understand meta-programs and types of analysis that are required to answer such questions, (2) flexible specification of breakpoints at which processing suspends, (3) supporting multiple ways of examining the processing state of a meta-program to understand the processing semantics and possible reasons of errors, and (4) collecting trace information for further examination upon termination of processing. All the proposed methods have been demonstrated within the analysis/debugging tool - IXP.

1.4 Thesis organization

The organization of this thesis is as follows:

In this chapter we have described the background of the research followed by the project objectives and scope, and contributions of this project.

Chapter 2 provides an overview of XVCL and the batch XVCL processor. This chapter describes basic concepts of XVCL and XVCL commands briefly, and explains how the batch XVCL processor works.

Chapter 3 describes some related work. This chapter introduces cxref, JSwat, Netron Fusion debugger and XSLT debugger. These dynamic analysis/debugging methods and tools can contribute some good ideas for our project.

Chapter 4 presents the research on the debugging meta-programs in XVCL.

Chapter 5 describes IXP in details. Readers can have insights in detailed features of IXP and how it is used from this chapter.

Chapter 6 summarizes the implementation of the batch XVCL processor. In this part, we introduce JDOM and provide the design concepts of the batch XVCL processor. This chapter mainly shows the detailed implementation of IXP. We focus on the design of the extended symbol table, DPSL (Debugging Points Specification Language), and four stepping functions. We also describe some other modifications made to the batch XVCL processor.

Chapter 7 concludes this thesis and suggests some possible future work.

Chapter 2

Overview of XVCL and XVCL processor

2.1 XVCL overview

2.1.1 What is an x-frame

When developing an XVCL solution, we partition a problem description (e.g. a software program) into x-frames. An x-frame is an XML (W3C, 1998) file containing reusable software asset instrumented with XVCL commands for change. Each x-frame contains a fragment of problem description, called Textual Content. The Textual Content is written in a base language, which can be a programming language such as Java, a natural language such as English or any other language [11] (XML is an exception. We may add XML support in the future).

XVCL commands are delimited by start (“<”) and end (“>”) tags, nesting properly within each other. Textual Content that is between end and start tags is instrumented with XVCL commands.

A typical x-frame is shown in Figure 2-1.

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file:///G:\XVCL\dtd\xvcl.dtd">

<x-frame name="A" language ="java">
        Texture Content 1
        <set var = "X" value = "5"/>
        <adapt x-frame = "B"/>
        Value of X is <value-of expr ="?@X?"/>
        Texture Content 2
</x-frame>
```

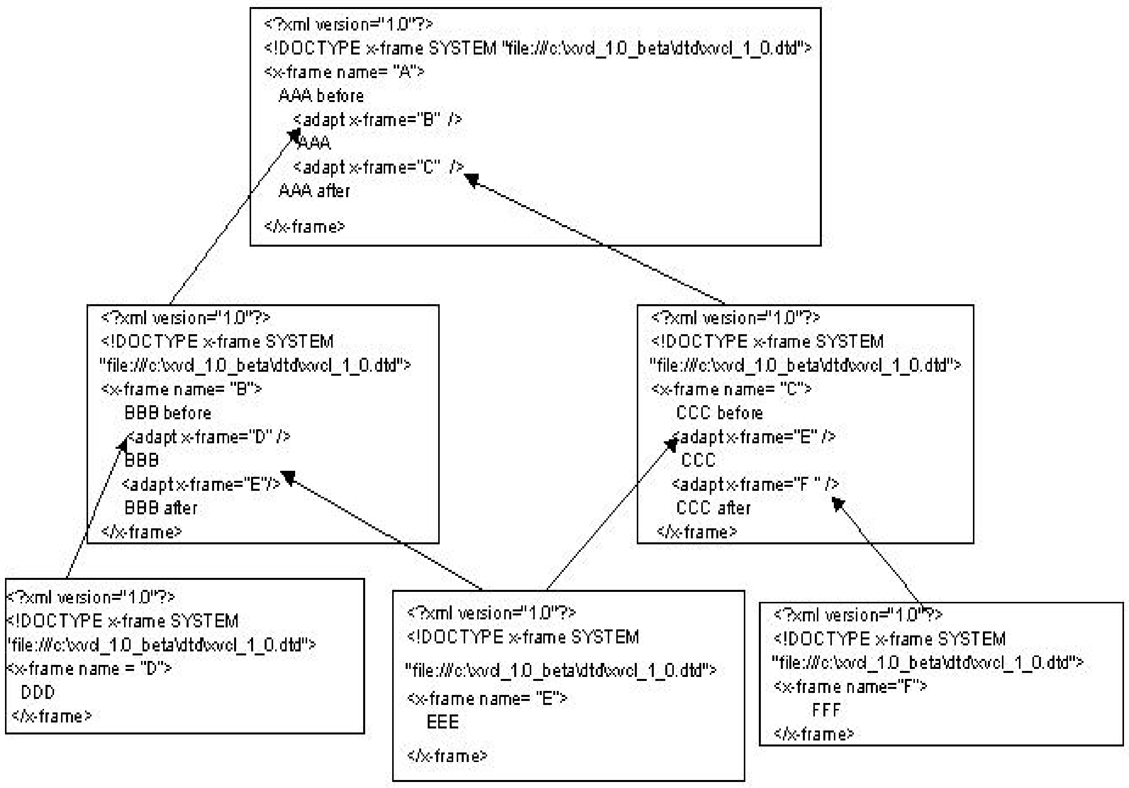
Figure 2-1 an example of an x-frame

2.1.2 An x-frame hierarchy

XVCL commands direct adaptation of x-frames. The x-frame adaptation process includes x-frame composition and customization. X-frames related by <adapt> commands forms a hierarchical structure, called x-framework. In the meta-language context, an x-frame is a meta-component, and an x-framework is an XVCL implementation of a meta-program component. The <adapt> command instructs the batch XVCL processor to process x-framework and emit the customized content of the adapted x-frame to the output. The root of the x-frame hierarchy is called the specification x-frame or SPC for short.

Different SPCs may yield different x-frame hierarchies. Figure 2-2 [16] illustrates an example of x-frame hierarchy.

- x-frame A adapts x-frames B and C
- x-frame B adapts D and E
- x-frame C adapts E and F
- x-frame A is the specification x-frame (SPC)



```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file://c:\xvcl_1.0_beta\dtd\xvcl_1_0.dtd">
<x-frame name= "A">
  AAA before
    <adapt x-frame="B" />
    AAA
    <adapt x-frame="C" />
  AAA after
</x-frame>
```

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM
"file://c:\xvcl_1.0_beta\dtd\xvcl_1_0.dtd">
<x-frame name= "B">
   BBB before
   <adapt x-frame="D" />
   BBB
   <adapt x-frame="E"/>
   BBB after
</x-frame>
```

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM
"file://c:\xvcl_1.0_beta\dtd\xvcl_1_0.dtd">
<x-frame name= "C">
   CCC before
   <adapt x-frame="E" />
    CCC
   <adapt x-frame="F " />
   CCC after
</x-frame>
```

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM
"file://c:\xvcl_1.0_beta\dtd\xvcl_1_0.dtd">
<x-frame name = "D">
  DDD
</x-frame>
```

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM
"file://c:\xvcl_1.0_beta\dtd\xvcl_1_0.dtd">
<x-frame name= "E">
  EEE
</x-frame>
```

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM
"file://c:\xvcl_1.0_beta\dtd\xvcl_1_0.dtd">
<x-frame name="F">
      FFF
</x-frame>
```

Figure 2-2 an example of x-frame hierarchy

2.2 XVCL commands

XVCL commands are expressed as XML tags [12]. These commands include <x-frame>, <break>, <adapt>, <insert>, <insert-before>, <insert-after>, <while>, <select>, <set>, <set-multi>, <value-of>, <message>, <remove>, <ifdef>, <ifndef>, <option>, <option-undefined>, <option-defined> and <otherwise>.

Figure 2-3 provides the current XVCL model that displays the whole set of XVCL commands, as well as their attributes.

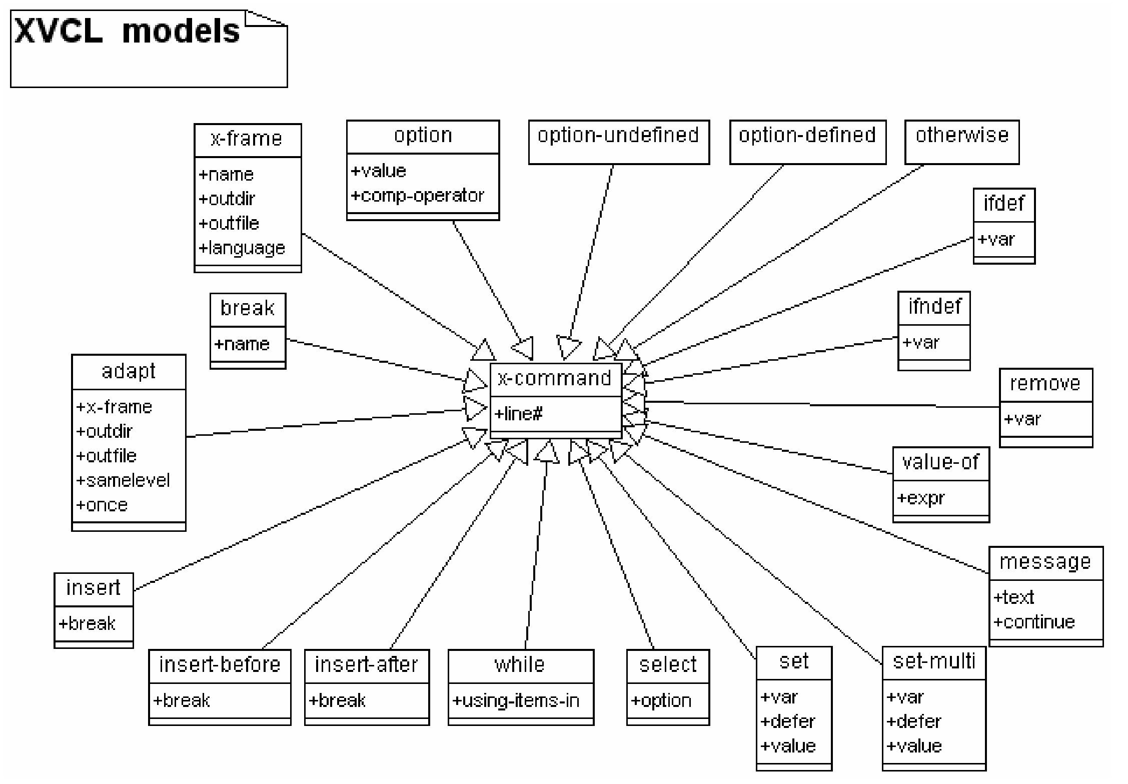


Figure 2-3 the current XVCL model

Each XVCL command must have its corresponding closing command (e.g. </adapt>). When there is no content between opening and closing x-command, the closing command can be replaced by a "/" symbol and attached at the end of opening command, like <adapt x-frame ="A"/>. XVCL commands are case sensitive. The letter case of opening command and closing command must be the same. The following example will not be recognized:

<adapt>   </ADAPT>

The validity checking of XVCL ensures an XVCL file obeys XVCL grammar rules. An XVCL file is valid if it conforms to the rules mentioned in its corresponding DTD (Data Type Definition). The DTD defines valid commands, their corresponding attributes, and nesting structure.

The DTD is case sensitive also. By default, all XVCL commands are in lower case letters.

In this section, we only describe several XVCL commands in details. In the Appendix B, readers can find summarized descriptions of all XVCL commands.

<x-frame>

Syntax

<**x-frame name** = “x-frame-name” [**outdir** = “dir-name”]

[**outfile** = “file-name”] [**language** = “language-name”]>

x-frame body: mixture of Textual Content and XVCL commands

</**x-frame**>

Command Definition

The <x-frame> command denotes the start and end of x-frame body in an x-frame. The x-frame body contains textual contents (e.g., program code, test cases, etc.) instrumented with XVCL commands for ease of adaptation.

Attribute Definition

name: name of the x-frame.

outdir: specifies a directory where the XVCL processor will store the file with the output emitted from the current x-frame.

outfile: specifies a file to which the XVCL processor will emit the output from the current x-frame.

language: specifies the base language in which the Textual Content of the x-frame is written (e.g., text, java, etc).

<adapt>

Syntax

<**adapt x-frame**="file-name" [**outdir**="dir-name"]

[**outfile**="file-name"] [**samelevel**="yes-no"] [**once**= "yes-no"]>

adapt-body: mixture of <insert>, <insert-before>, <insert-after> commands

</**adapt**>

Command Definition

The <adapt> command instructs the processor to:

- process the x-frame specified as the value of the "x-frame" attribute, "*file-name*",
- process all the descendent x-frames of the above x-frame,
- perform customizations of visited x-frames as specified in the body of the <adapt> command and
- emit the output to the specified output file

An x-frame is not allowed to <adapt> itself or any of its ancestor x-frames, i.e. recursive adaptations are not allowed.

Attribute Definition

x-frame: name of the x-frame to be adapted.

outdir: specifies a directory where the XVCL processor will store the file with the output emitted from the <adapt>ed x-frame.

outfile: specifies a file into which the XVCL processor will emit the output from the <adapt>ed x-frame.

samelevel: specifies whether or not to raise the variables declared in the adapted <x- frame> to the current x-frame (that is one that contains the <adapt> command).

once: specifies whether or not the subsequent <adapt>s of the <adapt>ed x-frame should be ignored (value "yes") or not (value "no").

<set>

Syntax

<**set var**="single-var-name" **value**="value" [**defer-evaluation**="yes-no"]/>

Command Definition

The <set> command is used to define a single-value variable. The <set> command assigns a "*value*" defined in "value" attribute to single-value variable "*var-name*" defined in "var" attribute.

Attribute Definition

var: name of the single-value variable.

value: value to be assigned.

defer-evaluation: specifies if the evaluation of the variable's value should be deferred (indicated by value "yes") or not (indicated by value "no"). If this attribute has value "*yes*", the value of that variable is evaluated at the reference point rather than at the point where the variable is <set>.

2.3 The batch XVCL processor overview

XVCL is more than a language for configuring variants. It is accompanied by a methodology and supported by a tool – the batch XVCL processor, which can automate the process of producing customized system from the specification.

X-framework processing involves customizing the contents of x-frames and assembling the results into organized output files. The whole processing is done by the batch XVCL processor. The XVCL processor traverses the x-frame hierarchy starting from the SPC adapts visited x-frames according to instructions of XVCL commands in SPC and outputs the result into one or more files. The following example aims to provide the understanding of the basic traversal processing structure of the XVCL processor. The processor's traversal order is dictated by <adapt> commands embedded in x-frames. Figure 2-4 [16] shows the traversal order of the XVCL processor for the x-frame hierarchy shown in Figure 2-2. In this example, the result is emitted into a single file and it shows on the right hand side of the Figure 2-4.

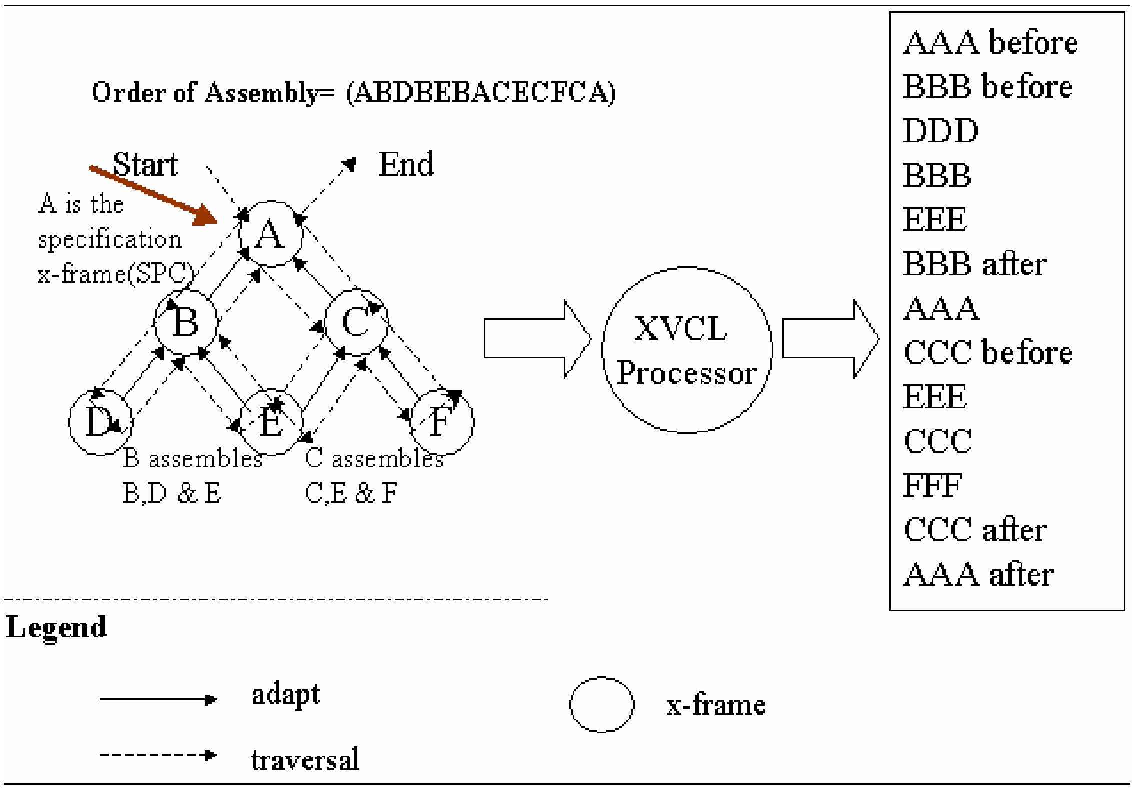


Figure 2-4 an example of the processing of an x-framework

Chapter 3

Related Work

In this chapter, we describe dynamic analysis/debugging methods and tools that are related to our project.

3.1 Cxref

Cxref [4] is a program that is used to generate C program cross-reference information based on static analysis of the collection of C program files. The cxref command analyzes a collection of C program files and builds a cross-reference table. The table includes four fields: NAME, FILE, FUNCTION, and LINE. The line numbers appearing in the LINE field also show reference marks as appropriate. The reference marks include:

assignment	=
declaration	-
definition	*

In cxref, the cross referencing is performed for the following program items [4]:

- Files
    - ➢ The files that the current file is included in (even when included via other files).
- Variables
    - ➢ The location of the definition of external variables.
    - ➢ The files that have visibility of global variables.
    - ➢ The files / functions that use the variable.

- Functions

   - The file that the function is prototyped in.
   - The functions that the function calls.
   - The functions that call the function.
   - The files and functions that reference the function.
   - The variables that are used in the function.

The cxref shows the relationships among of the above program items. The following example illustrates information produced by cxref.

test.c

```
1.  int add(int, int);
2.  main() {
3.          int i = 0;
4.          int j = 1;
5.          add (i, j);
6.  }
7.
8.  int add (int i, int j) {
9.        return i+j;
10. }
```

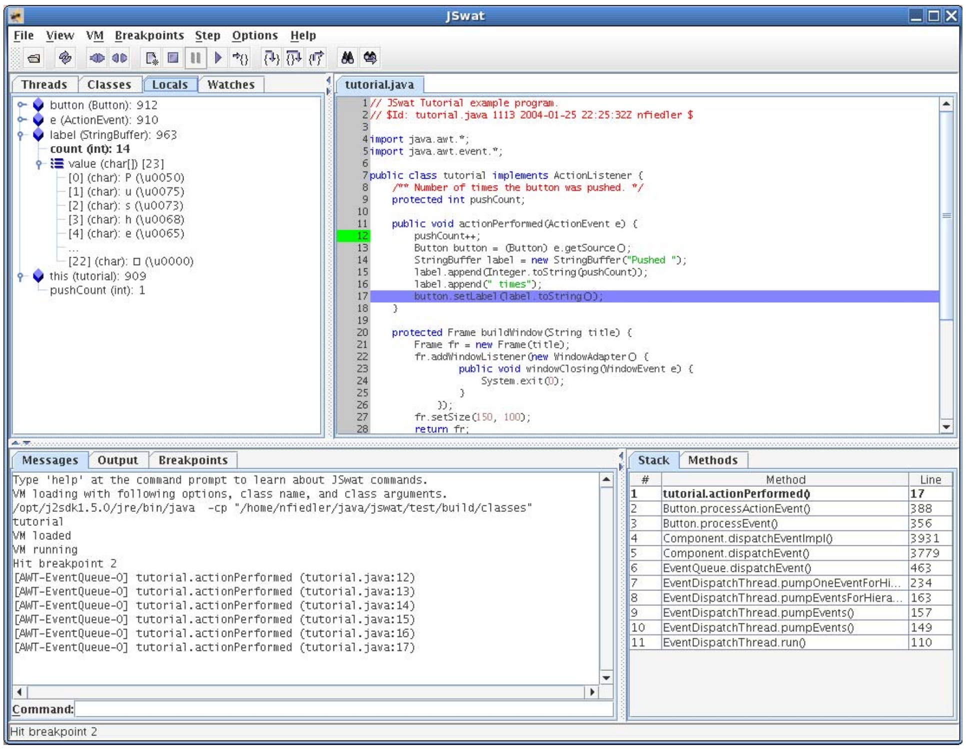
The resulting cross-reference table is:

```
test.c:
NAME        FILE       FUNCTION      LINE
__func__    test.c      add            8*
__func__    test.c      main           2*
add          test.c      ---          1-     5     8*
i           test.c      add          8*     9
i           test.c      main         3*     3=    5
j           test.c      add          8*     9
j           test.c      main         4*     4=    5
main         test.c      ---           2*
```

Like cxref, our IXP also collects cross-reference information about meta-program items. However in case of meta-programs, not much useful information can be collected in the course of static program analysis. Therefore, unlike cxref, IXP does both static and dynamic analysis to produce cross-reference information and meta-program processing traces.

3.2 JSwat

JSwat is a standalone, graphical Java debugger [7].


JSwat
File View VM Breakpoints Step Options Help
Threads Classes Locals Watches
tutorial.java
Messages Output Breakpoints
Stack Methods
Command:

```
button (Button): 912
e (ActionEvent): 910
label (StringBuffer): 963
    count (int): 14
    value (char[]) [23]
        [0] (char): P (\u0050)
        [1] (char): u (\u0075)
        [2] (char): s (\u0073)
        [3] (char): h (\u0068)
        [4] (char): e (\u0065)
        ...
        [22] (char): □ (\u0000)
this (tutorial): 909
    pushCount (int): 1
```

```java
// JSwat Tutorial example program.
// $Id: tutorial.java 1113 2004-01-25 22:25:32Z nfiedler $

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

public class tutorial implements ActionListener {
    /** Number of times the button was pushed. */
    protected int pushCount;

    public void actionPerformed(ActionEvent e) {
        pushCount++;
        Button button = (Button) e.getSource();
        StringBuffer label = new StringBuffer("Pushed ");
        label.append(Integer.toString(pushCount));
        label.append(" times");
        button.setLabel(label.toString());
    }

    protected Frame buildWindow(String title) {
        Frame fr = new Frame(title);
        fr.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });
        fr.setSize(150, 100);
        return fr;
```

```
Type 'help' at the command prompt to learn about JSwat commands.
VM loading with following options, class name, and class arguments.
/opt/j2sdk1.5.0/jre/bin/java  -cp "/home/nfiedler/java/jswat/test/build/classes"
tutorial
VM loaded
VM running
Hit breakpoint 2
[AWT-EventQueue-0] tutorial.actionPerformed (tutorial.java:12)
[AWT-EventQueue-0] tutorial.actionPerformed (tutorial.java:13)
[AWT-EventQueue-0] tutorial.actionPerformed (tutorial.java:14)
[AWT-EventQueue-0] tutorial.actionPerformed (tutorial.java:15)
[AWT-EventQueue-0] tutorial.actionPerformed (tutorial.java:16)
[AWT-EventQueue-0] tutorial.actionPerformed (tutorial.java:17)
```

#	Method	Line
1	**tutorial.actionPerformed()**	**17**
2	Button.processActionEvent()	388
3	Button.processEvent()	356
4	Component.dispatchEventImpl()	3931
5	Component.dispatchEvent()	3779
6	EventQueue.dispatchEvent()	463
7	EventDispatchThread.pumpOneEventForHi...	234
8	EventDispatchThread.pumpEventsForHiera...	163
9	EventDispatchThread.pumpEvents()	157
10	EventDispatchThread.pumpEvents()	149
11	EventDispatchThread.run()	110

Hit breakpoint 2

Figure 3-1 JSwat – a graphical java debugger

Its features include:

- setting breakpoints
- single-stepping
- colorized source code display
- watching stack frames and variables
- a non-graphical mode for use in consoles

The above features are commonly found in debuggers of programming-languages.

3.3 Netron Fusion

3.3.1 What is Netron Fusion

Netron Fusion™ delivers powerful support for component-based development, with flexible application frameworks, proven reusable components, and robust development and diagnostic tools [9].

With Netron Fusion™, the user can harness the power of software reuse and quickly build complex business systems for any architecture. The approach to software reuse has been shown to produce core systems 70% faster than the industry average [9].

Fusion's component-based approach lets the user separate standards and business rules from the technical architecture, making it easier and faster to move existing systems forward while collapsing maintenance effort.

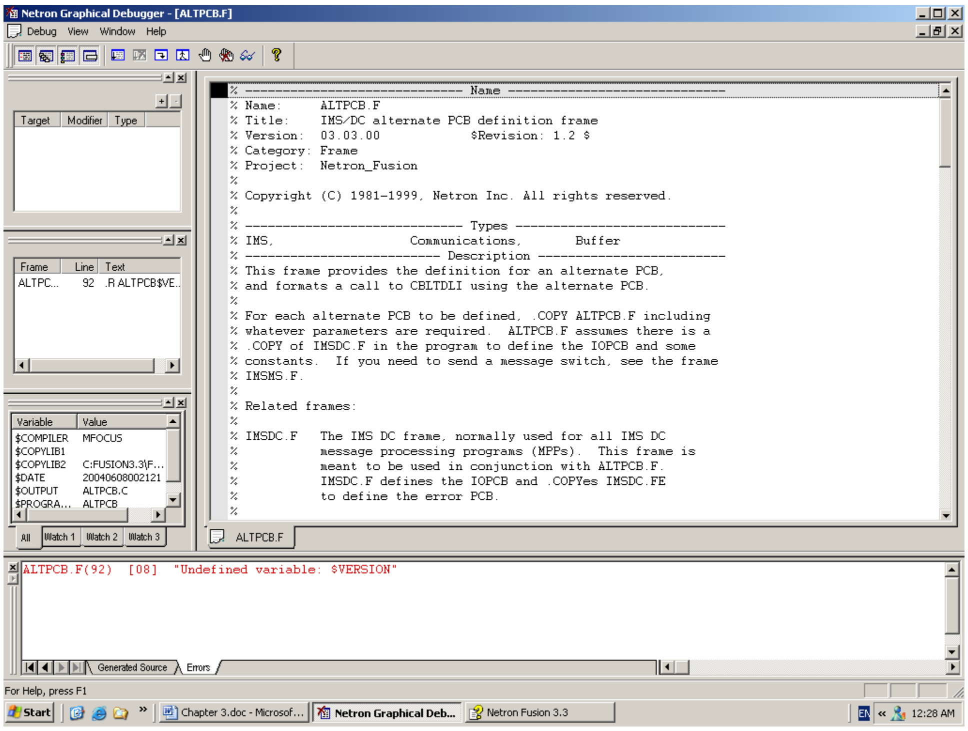
Netron Fusion is based on the frame technology and it is a meta-programming language. The following shows an example of Netron Fusion frame.

```
;ALTPCB-SEND-01.
.BREAK ;ALTPCB-SEND
.SELECT ;ALTPCBTYPE
.WHEN MODIFIABLE
      CALL "CBLTDLI" USING
           ;IMSDC-CHNG
           ;ALTPCB
           ;ALTPCB-DESTINATION.
      IF ;ALTPCB-STATUS NOT = SPACE ;ALTPCBERRORACTION.
.END-SELECT
```

```
         CALL "CBLTDLI" USING
                ;IMSDC-ISRT
                ;ALTPCB
                ;IMSDC-IOREC
.SELECT ;STINGRAY
.WHEN <>
. SELECT ;ALTPCBTYPE
. WHEN MODIFIABLE
                ;ALTPCB-MODNAME
. END-SELECT ;ALTPCBTYPE
.END-SELECT ;STINGRAY
  .IF ;ALTPCB-STATUS NOT = SPACE ;ALTPCBERRORACTION.
.SELECT ;ALTPCBTYPE
.WHEN MODIFIABLE
         CALL "CBLTDLI" USING
                ;IMSDC-PURG
                ;ALTPCB.
         IF ;ALTPCB-STATUS NOT = SPACE ;ALTPCBERRORACTION.
.END-SELECT
.END-BREAK ;ALTPCB-SEND
%
.BREAK ;ALTPCB-ALTPCB.F-NEW-CODE
.END-BREAK ;ALTPCB-ALTPCB.F-NEW-CODE
```

3.3.2 Netron Fusion debugger

Netron processor includes a graphical debugger that allows the user to detect processing errors in the program quickly and easily. The debugger lets the user process at any speed, giving the user a line-by-line perspective of the operation. The user can visually monitor the values of the variables as they change throughout the program, and force Netron processor to pause on a specific string, line, or variable. In addition, the user can see how and when lines of text are emitted after command evaluation. The debugger is not only useful if there are processing errors in the application; it is also useful if the application is processed successfully, but does not behave as the user expected.



Target	Modifier	Type

Frame	Line	Text
ALTPC...	92	.R ALTPCB$VE...

Variable	Value
$COMPILER	MFOCUS
$COPYLIB1	
$COPYLIB2	C:FUSION3.3\F...
$DATE	20040608002121
$OUTPUT	ALTPCB.C
$PROGRA...	ALTPCB

```
% ----------------------------- Name -----------------------------
% Name:     ALTPCB.F
% Title:    IMS/DC alternate PCB definition frame
% Version:  03.03.00            $Revision: 1.2 $
% Category: Frame
% Project:  Netron_Fusion
%
% Copyright (C) 1981-1999, Netron Inc. All rights reserved.
%
% ----------------------------- Types -----------------------------
% IMS,                  Communications,       Buffer
% -------------------------- Description --------------------------
% This frame provides the definition for an alternate PCB,
% and formats a call to CBLTDLI using the alternate PCB.
%
% For each alternate PCB to be defined, .COPY ALTPCB.F including
% whatever parameters are required.  ALTPCB.F assumes there is a
% .COPY of IMSDC.F in the program to define the IOPCB and some
% constants.  If you need to send a message switch, see the frame
% IMSMS.F.
%
% Related frames:
%
% IMSDC.F   The IMS DC frame, normally used for all IMS DC
%           message processing programs (MPPs).  This frame is
%           meant to be used in conjunction with ALTPCB.F.
%           IMSDC.F defines the IOPCB and .COPYes IMSDC.FE
%           to define the error PCB.
%
```

```
ALTPCB.F(92)  [08]  "Undefined variable: $VERSION"
```

Figure 3-2 Netron Fusion graphical debugger

The following brief list describes some common debugging tasks.

- Control the Netron processor

From the Graphical debugger, there are several functions available that allow the user to control the Netron processor. They are:

  ➢ Start

    Start instructs the Netron processor to start processing the frame source, pausing only if halted by the debugger (see Halt), if an error is encountered, or the end of the frame source is reached.

- ➢ Stop

  Stopping or suspending the Netron processor causes it to stop processing the frame source and wait until further instructions are given by the debugger. This operation is only available while a continue operation is in effect.

- ➢ Step

  To "Step the Netron processor" means to instruct it to process a given number of lines and then return control to the debugger, to await further instructions. Note that the step operation ignores breakpoints.

- ➢ Walk

  Walking consists of a series of single steps. The walk operation ignores breakpoints and will walk right past a breakpoint without stopping.

- View a frame stack entry

The frame stack consists of a set of one or more entries that describe frames that the Netron processor currently has open.

- Set and remove a breakpoint

The result of the breakpoint addition and deletion will be displayed in the Breakpoint Window.

- Monitor variables

To monitor the value of a variable or compound variable, the user must add a *watch*. That is, the user creates an entry in one of the Watch Window for the variable in question.

- Change variable values

While the Netron processor is paused, it is possible to modify the current value of any of the variables that have been defined so far. To do so, choose Edit Variable from the Variable Window context menu. This displays the Edit Variable Dialog.

3.4 XSLT

3.4.1 What is XSLT

XSLT stands XSL (eXtensible Stylesheet Language) transformations [15]. It is designed for use as part of XSL, which is a stylesheet language for XML. XSLT is a language for transforming XML documents into other XML documents. Based on this, XSLT can be regarded as a meta-language. The following is a small example, which shows how XSLT transforms an XML document to another XML document.

The original XML document is:

```
<card type="simple">
   <name>John Doe</name>
   <title>CEO, Widget Inc.</title>
   <email>john.doe@widget.com</email>
   <phone>(202) 456-1414</phone>
</card>
```

The XSLT stylesheet is:

```
<xsl:stylesheet       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0"
              xmlns="http://www.w3.org/1999/xhtml">


   <xsl:template match="card[@type='simple']">
     <html xmlns="http://www.w3.org/1999/xhtml">
       <title>business card</title><body>
```

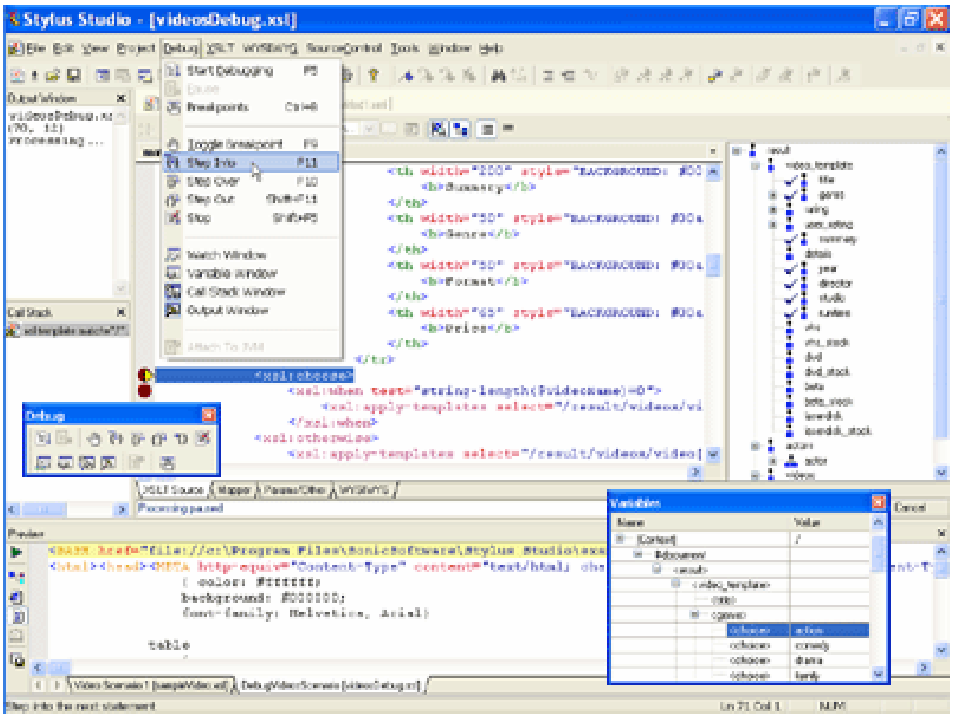
```
        <xsl:apply-templates select="name"/>
        <xsl:apply-templates select="title"/>
        <xsl:apply-templates select="email"/>
        <xsl:apply-templates select="phone"/>
    </body></html>
  </xsl:template>

  <xsl:template match="card/name">
    <h1><xsl:value-of select="text()"/></h1>
  </xsl:template>

  <xsl:template match="email">
    <p>email: <a href="mailto:{text()}"><tt>
      <xsl:value-of select="text()"/>
    </tt></a></p>
  </xsl:template>
</xsl:stylesheet>
```

After transformation, we get another XML document:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<title>business card</title>
<body><h1>John Doe</h1><h3><i>CEO, Widget Inc.</i></h3>
<p> email: <ahref="mailto:john.doe@widget.com">
<tt>john.doe@widget.com</tt></a></p>
<p>phone: (202) 456-1414</p>
</body></html>
```

3.4.2 XSLT debugger

Stylus Studio's XSLT debugger [13] is one of the most popular XSLT debuggers. It gives the user complete visibility and control over the XSLT transformation process.



```
<th width="200" style="BACKGROUND: #00
    <b>Summary</b>
</th>
<th width="50" style="BACKGROUND: #00a
    <b>Genre</b>
</th>
<th width="50" style="BACKGROUND: #00a
    <b>Format</b>
</th>
<th width="65" style="BACKGROUND: #00a
    <b>Price</b>
</th>
</tr>
<xsl:choose>
    <xsl:when test="string-length($videoname)=0">
        <xsl:apply-templates select="/result/videos/vi
    </xsl:when>
<xsl:otherwise>
    <xsl:apply-templates select="/result/videos/video[
```

```
<BASE href="file://c:\Program Files\SonicSoftware\Stylus Studio\exa
<html><head><META http-equiv="Content-Type" content="text/html; cha
                { color: #ffffff;
                background: #000000;
                font-family: Helvetica, Arial}

            table
```

Figure 3-3 Stylus Studio's XSLT debugger

It comes with the following functionalities:

- Set/Toggle XSLT Breakpoints

XSLT debugger allows the user to easily set and toggle breakpoints anywhere in XSLT stylesheet code.

- Single Step-Through XSLT Stylesheets

Using Stylus Studio's XSLT Debugger, the user can step through the execution of an XSLT stylesheet, line-by-line, just as the user would using a conventional software debugger. Using the XSLT Debugger toolbar, the user can:

  - ➢ Step-In: Execute the next line of XSLT stylesheet code.
  - ➢ Step-Out: Return to the line of XSLT code that called the template the user is in.

  - Step-Over: Execute the XSLT template being called, and return immediately.

- Inspect XSLT Variables

Stylus Studio's XSLT debugger has an XSLT Variables window that shows all of the XSLT variables and values that are in scope, relative to the XSLT processor's current context. The contents of the XSLT Variables window change dynamically as the user steps through the XSLT transformation process.

- Set Watches on XSLT Variables or XPath Expressions

Similarly, in addition to viewing all of the XSLT variables in scope, the user can set watches on custom XSLT variables or any XPath expression. Because variables and XPath expressions are typically themselves tree fragments, the user can easily expand/collapse them in the Watch window.

- XSLT Output Window

Stylus Studio's XSLT debugger provides a XSLT Output window that displays the output of the XSLT processor in real time, as the user walks through the XSLT transformation process. One of the helpful features included in the XSLT Output window is Backmapping — if the user clicks on a line of output, Stylus Studio highlights the XSLT stylesheet source that generated that line of output. Backmapping is an invaluable tool for troubleshooting incorrect XSLT stylesheet output, and for answering the age-old question: "How did *that* get there?" Other features include the ability to refresh the output, save the output to a file, compare different XSLT outputs, and much more.

Chapter 4

The Research on the debugging of XVCL programs

XVCL is a meta-language. By definition, meta-programming is a method to develop computer programs. It works by generating source code in a target language from a program specification in a higher level language.

XVCL program source code is based on XML. The XVCL processor can be used to generate the program from XVCL source code to potentially any target language, such as Java, C, or pure English language. The support for XML can be added in the future.

Meta-language is different from the programming language (like Java, C, and C++). The meta-programs are not directly executable - they must be instantiated at the construction time to produce an executable program. Therefore, debugging of meta-programs is a different issue compared with debugging of programs in programming-language. And it is almost a virtually unknown area.

In this chapter, we do the research on the debugging of meta-programs in XVCL. At the first step, we should get a clear idea on what the debugging is.

4.1 What is debugging

4.1.1 Debugging objective

The objective of debugging is to find and correct the cause of a software error [1].

Debugging is part of testing. When a test case uncovers an error, debugging is the process that results in the removal of the error. A programmer, evaluating the results of a test, is often confronted with a symptomatic indication of an error. Sometimes there is no obvious relationship between the external manifestation of the error and its internal cause. Consequently, debugging is often described as an art.

4.1.2 Debugging process

The debugging process begins with the execution of a test case (Figure 4-1) [1] and it attempts to match symptom with cause, thereby leading to error correction.

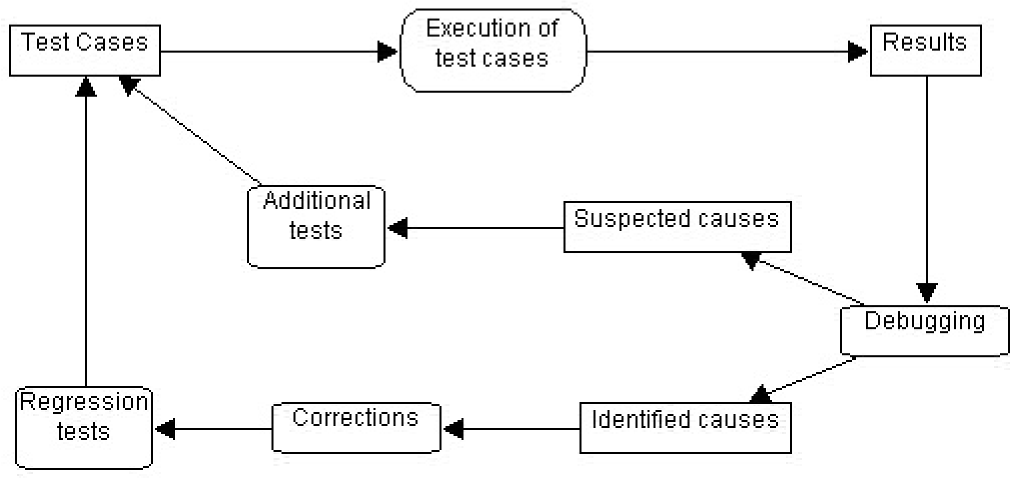


Figure 4-1 debugging process

4.1.3 Debugging approaches

In general, there are three debugging approaches [1].

- Brute force

This is the most common approach and also the least effective method for isolating the cause of a software error. This method is usually applied when all else fails. In this approach, a program is instrumented with write statements and/or memory dumps are used with the hope that a clue to the cause of the error may be found in the large amount of information collected.

- Backtracking

This is a fairly common approach that is especially successful in small programs. Beginning at the program point where a symptom has been uncovered, the source code is traced backwards manually until the cause is found. Unfortunately as the program size increases, the number of potential backward paths may become unmanageably large.

- Cause elimination

This approach uses induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organised to isolate potential causes. A cause hypothesis is devised and the above data are used to prove or disprove the hypothesis. Alternatively, a list of all the possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

4.2 Problems with understanding XVCL

4.2.1 Possible errors of meta-programs in XVCL

In meta-programs written in XVCL, there are two types of possible errors. The first type of error is fatal error, which leads the processor to abnormal termination of the processing. This happens, for example, when a variable is undefined or an adapted x-frame is missing. In both cases, the batch XVCL processor reports the error to the user and terminates abnormally. These errors often occur in x-frameworks under development and they can be fixed easily by users. The second type of error does not cause the XVCL processor to fail. The processor executes the x-frameworks successfully. But the generated custom program is buggy or it does not behave as the user expects. In such a case, two sources must be examined - meta-programs written in XVCL, and the programs generated by XVCL. Debugging programs (Java, C, C++ and etc) generated by XVCL is another story. It is beyond the scope of our project. Here, we only address the errors which come from the meta-programs in XVCL. To trace these errors, we need a tool support for debugging. IXP serves this purpose.

4.2.2 Comparison between debugging of C programs and debugging of XVCL programs

In this section, we compare the debugging of XVCL programs with the debugging of C programs

Although these two languages are quite different, when we debug programs written in them, there are some similarities. In both cases, debugging includes:

- Breakpoint

  The user can set a breakpoint, which tells the program to stop executing at a specific line of code. He can also remove this breakpoint later during debugging.

- Step functions

  The user can step through the source codes line by line during the debugging. Stepping into (stepping out, stepping over) a called function in a C program is similar to stepping through (stepping out, stepping over) an adapted x-frame in an XVCL program.

- Watching variables

  The user can watch the values of variables change during debugging

- Modifying variables

The user can modify the values of variables when the program suspends.

Due to the difference between meta-language and programming-language, the debugging of XVCL programs has some special characteristics which are as follows.

- There are two ways to specify a breakpoint in the debugging of XVCL programs. One way is to tell the debugger to suspend processing at the specific XVCL command in an x-frame (e.g. suspend at the line 6 of x-frame A). Another way is to tell the debugger to suspend processing at the specific type of XVCL commands in a list of x-frames (e.g. suspend at <adapt> commands in x-frame A and B). Readers can get the details from the section 5.5.3 Setting debugging points.

- Step out function is more complicated. In step out mode, the execution can also exit from any XVCL structures (<while> and <select>). Please refer to the section 5.5.8 Single-stepping.

- During the debugging of XVCL programs, the user can examine the partially emitted outputs. Therefore, the user can get a clear idea on which part of codes contribute to which part of outputs.

- There are two options when the user tries to modify the value of a variable. This is due to the variable scoping rule of XVCL. Readers can get the details from the section 5.5.6 Modifying values of variables when the processing suspends.

4.3 Requirements for XVCL analysis and debugging

4.3.1 Difficulties and the process of formulating requirements

- Interviews and observations from other related works

We did studies in related areas, as described in the Chapter 3. Based on the interviews and observations from related works, we formulated basic requirements of our debugging tool. Features like “single-stepping”, “collecting cross-reference information”, “watching and modifying variables”, and “setting a breakpoint” are basic and crucial in other debuggers. These features were also deemed to be useful in our tool.

However, XVCL is a meta-programming system. It is different from programming languages in many concepts. The very execution process and emitting outputs are different from conventional programs. Therefore, the requirements for XVCL analysis and debugging could not be formulated directly based on the existing tools. For instance, Cxref can collect static cross-reference information of C programs only. But since XVCL is a meta-language, not much useful information can be collected in the course of static program analysis. We need to collect both static and dynamic analysis to produce the meta-program processing trace.

Formulation of the requirements for XVCL analysis and debugging was an interactive process that required much imagination and effort.

- Brainstorming with SES System and our team members

Our reuse group has regular meetings with SES System Pte Ltd every two weeks. During the meetings, we discussed the analysis and debugging features of XVCL. As users of XVCL, SES staffs proposed some required features. For instance, Mr. Ulf Pettersson, SES team leader, suggested us to provide the user more trace information such as the user's relevant action in case of fatal errors. We thought it was a good idea. We have embedded this feature in our tool. We also demoed the tool during some meetings. We got feedbacks from SES system on the GUI design and the features of the tool. We evaluated these feedbacks and improved our design and implementation accordingly.

We also shared ideas among our team. We discussed the essential features to fulfill the analysis and debugging requirement. Some valuable suggestions like "giving the user options for the lifescope of a variable modification", "Step Out function can also be applied to <while> and <select> XVCL commands" and etc have been proposed by our team members. They also tried the tool by themselves. They provided feedbacks and suggested improvements on the design of the tool.

- Prototyping

To early demonstrate and test requirements, we applied rapid prototyping in the project. We use UML use case diagrams to discuss requirements. Based on the diagrams, we could get a clear idea of what the user could do with this tool. We also constructed the state chart diagram of the tool (see Figure 5-1). With the help of this diagram, we could know exactly the life cycle of our tool. The prototyping, in ways of developing UML models, made our design clearer and made our implementation easier.

Chapter 5

Description of the interactive XVCL processor

5.1 Why interactive XVCL processor

The batch XVCL processor works well when everything goes fine during custom program generation. However, it may not play its role well when something goes wrong – there may be an error either in an x-framework or in the way the user specifies the system. For example, if the batch XVCL processor encounters a reference to an undefined variable "x" during processing of an x-framework, it only reports the error "x is undefined" to the user and terminates the processing abnormally. However, that may not be what the user expects. The user may want to find the source of the error and even remedy the error when it is spotted. Another example is given. The user assigns a wrong value to a variable by mistake when he customizes the system. After the batch XVCL processor completes the processing of the system, the user gets a result which is different from what he expects. To find the source of the error, the user needs trace and understand the processing sequence. It is quite difficult without tool support.

In this project, we designed an interactive, debugging tool – IXP (Interactive XVCL Processor) for XVCL. IXP helps the user analyze processing of x-frameworks and provides debugging environment for XVCL. IXP is based on the batch XVCL processor. It inherits all functionalities of the batch XVCL processor. In addition, IXP also owns some extended functionalities that the batch XVCL processor does not have. We will describe these functionalities particularly in section 5.5.

5.2 How to run interactive XVCL processor

The programmer uses the batch XVCL processor to process an x-framework (run command “java xvcl.XVCL ‘SPC name’” in the command line). In case that the processor encounters an error or the generated result does not behave as the user expects, the user can start the interactive XVCL processor with opening the same SPC (run command “java ixp.XVCL ‘SPC name’” in the command line). With the help of IXP, the user can do the debugging of the XVCL program.

5.3 How interactive XVCL processor works

IXP is based on the batch XVCL processor. The remarkable difference is that IXP may suspend the processing of an x-framework in some situations. IXP suspends x-framework processing whenever it encounters an error (e.g. a reference to an undefined variable) or it reaches a user’s designated “debugging point”. Here, a debugging point refers to an XVCL command, at which the user expects the processing of an x-framework to suspend (The counterpart of a debugging point in a C debugger is a breakpoint. To differentiate breakpoints set by <break> x-commands, in XVCL context, this point is called “debugging point” instead). For detailed description of the debugging point, readers may refer to section 6.3.2 Debugging Points Specification Language (DPSL). Figure 5-1 shows the state chart diagram of IXP.

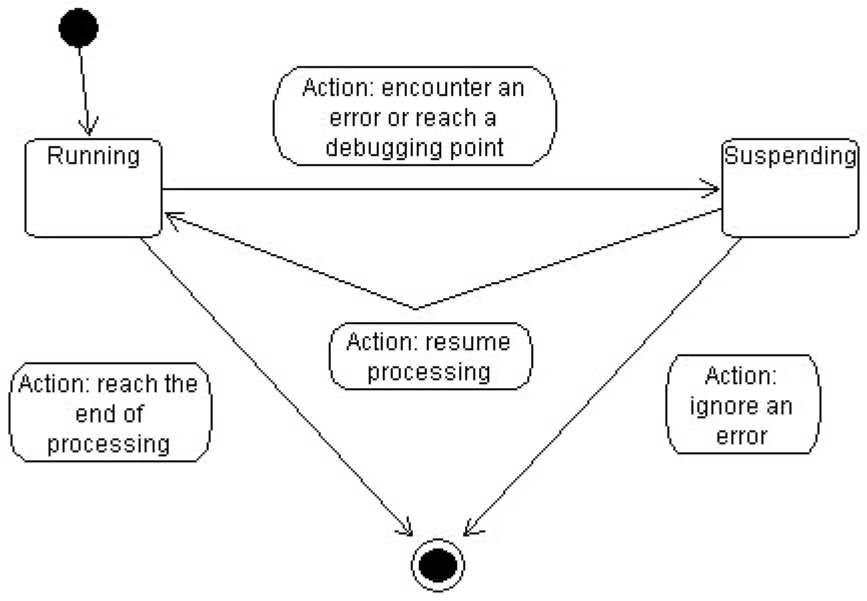

Action: encounter an error or reach a debugging point
Running
Suspending
Action: resume processing
Action: reach the end of processing
Action: ignore an error

Figure 5-1 the state chart diagram of IXP

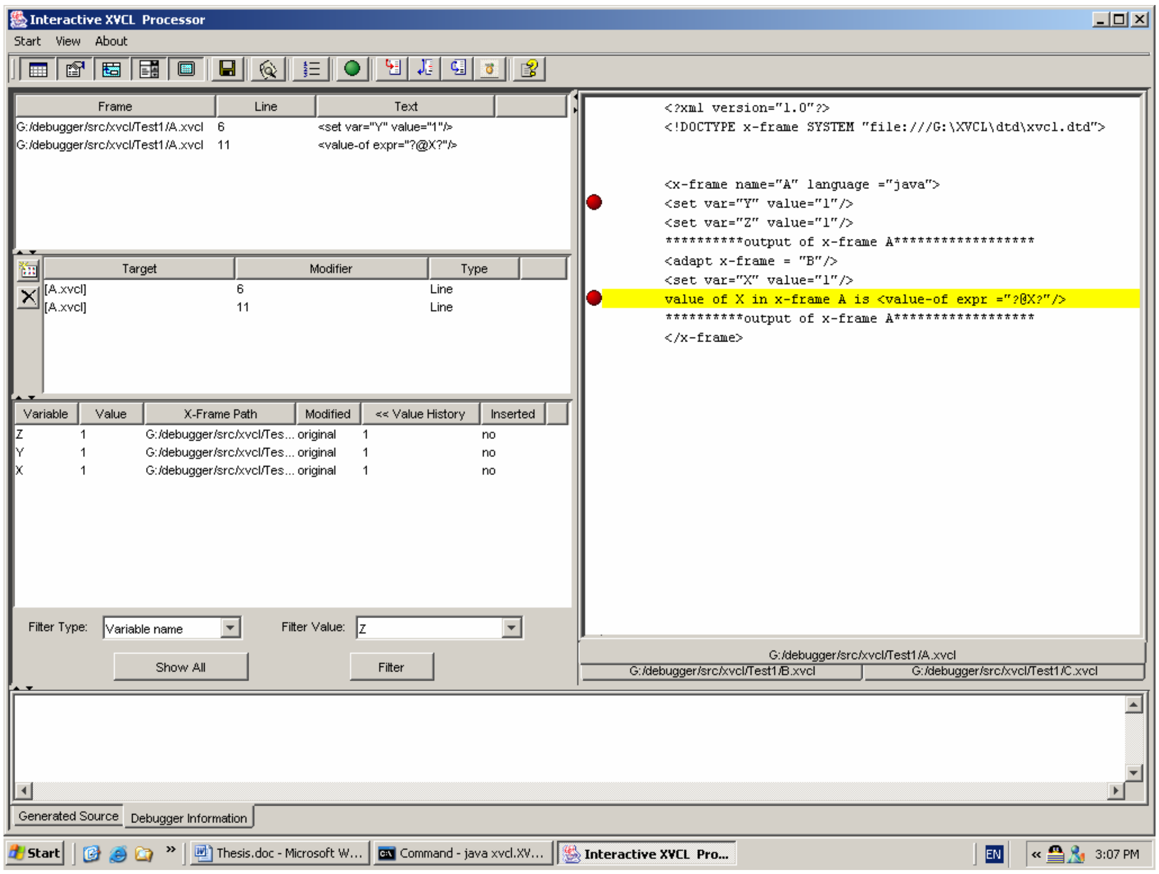
When processing of an x-framework suspends, the user may do the following:

- feed a value to an undefined variable, or designate a frame to an undefined adapted x-frame
- set extra debugging points
- remove certain existing debugging points
- set an end_debugging point (an end_debugging point instructs IXP to ignore any future debugging point in the subsequent processing)
- modify values of some variables
- save the trace information
- set the output formatting
- change the stepping mode (e.g. step through -> step out)
- examine the partial output emitted

After that, the user may resume the processing of an x-framework. All modifications made during this suspending will be recorded by IXP and take effect in the follow-up processing. That is, IXP will process the x-framework in such a simulated environment. For example, the user modifies the value of variable “x” valued 10 originally, to 5 when the processing suspends. After the processing resumes, when IXP encounters a reference to “x” again, it knows the value of “x” is 5, not 10 any more.

5.4 An Overview of IXP’s GUI

The figure below shows the GUI of IXP. On the right-hand-side there is a Source Window that shows currently processed x-frame, with the current processing command highlighted. On the left-hand-side, we see the following windows (from the top).


Interactive XVCL Processor
Start
View
About
Filter Type:
Variable name
Filter Value:
Z
Show All
Filter
G:/debugger/src/xvcl/Test1/A.xvcl
G:/debugger/src/xvcl/Test1/B.xvcl
G:/debugger/src/xvcl/Test1/C.xvcl
Generated Source
Debugger Information
Start
Thesis.doc - Microsoft W...
Command - java xvcl.XV...
Interactive XVCL Pro...
EN
3:07 PM

Frame	Line	Text
G:/debugger/src/xvcl/Test1/A.xvcl	6	<set var="Y" value="1"/>
G:/debugger/src/xvcl/Test1/A.xvcl	11	<value-of expr="?@X?"/>

Target	Modifier	Type
[A.xvcl]	6	Line
[A.xvcl]	11	Line

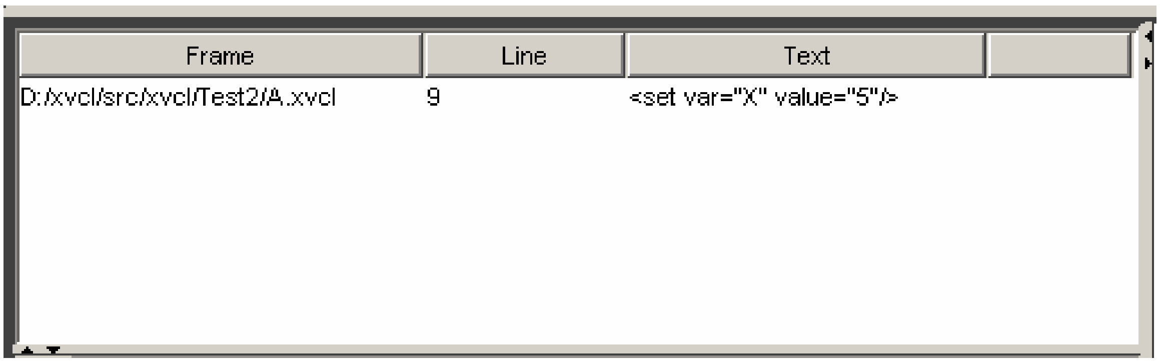
Variable	Value	X-Frame Path	Modified	<< Value History	Inserted
Z	1	G:/debugger/src/xvcl/Tes...	original	1	no
Y	1	G:/debugger/src/xvcl/Tes...	original	1	no
X	1	G:/debugger/src/xvcl/Tes...	original	1	no

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file:///G:\XVCL\dtd\xvcl.dtd">


<x-frame name="A" language ="java">
<set var="Y" value="1"/>
<set var="Z" value="1"/>
**********output of x-frame A*******************
<adapt x-frame = "B"/>
<set var="X" value="1"/>
value of X in x-frame A is <value-of expr ="?@X?"/>
**********output of x-frame A*******************
</x-frame>
```

Figure 5-2 GUI of IXP

5.4.1 Target Window

The Target Window shows all currently open frames, and lines at which the processing has been suspended.



Frame	Line	Text	
D:/xvcl/src/xvcl/Test2/A.xvcl	9	<set var="X" value="5"/>	

Figure 5-3 the Target Window

5.4.2 Debugging Point Window

The Debugging Point Window displays all the debugging points that are currently set. A debugging point description consists of the type of the debugging point, the target of the debugging point (i.e. in which x-frames), and the content of the debugging point (i.e., the modifier).

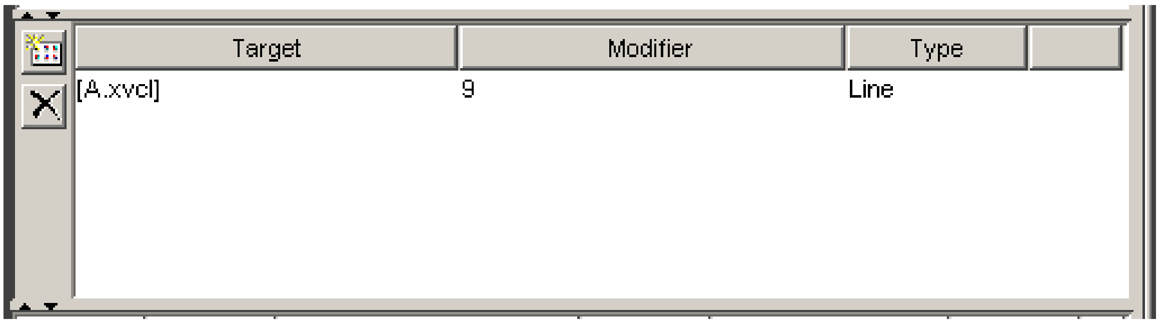
For convenience, the Debugging Point Window not only shows all the debugging points which are set by the user, but also allows the user to add, remove and modify these debugging points.

There are two buttons on the left margin of the Debugging Point Window.

The first button is used to set a debugging point.

The second button is used to delete a selected debugging point.

The user may select a debugging point and double click it. Then the user can modify the content of the selected debugging point. Please note, the modified debugging point must follow the syntax and semantic of DPSL (Debugging Point Specification Language, refer to Appendix).



Target	Modifier	Type	
[A.xvcl]	9	Line	

Figure 5-4 the Debugging Point Window

5.4.3 Variable Window

The Variable Window provides a view of the variables which are defined in the current adapting chain (*Current adapting chain* contains current processing x-frame and its ancestral x-frames. We can use an example to illustrate the definition of adapting chain. Given the x-frame hierarchy shown in Figure 2-2, when processing of x-framework suspends at x-frame D, the current adapting chain contains x-frame A, B, and D. When processing of x-framework suspends at x-frame F, the current adapting chain contains x-frame A, C, and F).

The information about a variable includes: variable's name, current variable's value, variable's x-frame path (in which x-frame the variable is defined), modified state (whether this variable has been modified or not), variables' value list (a list of values that has been assigned to this variable, current value is placed at the head of this list) and inserted state (whether this variable is newly inserted or not).

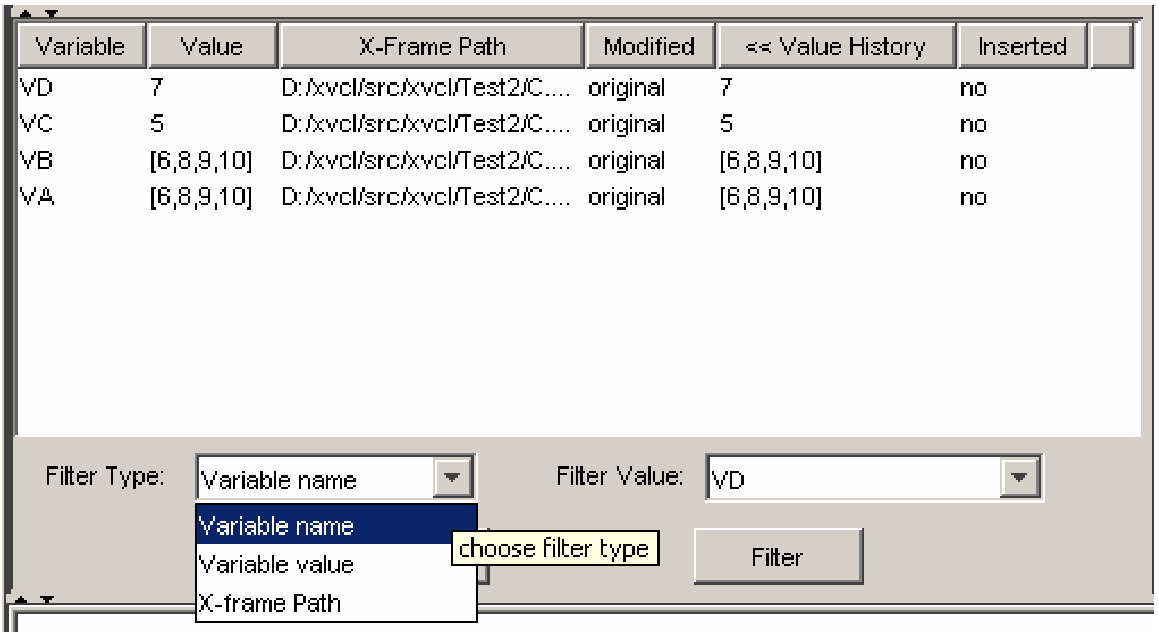
In the Variable Window, the user can use a context menu to edit the value of a variable, and restore the value of a variable to its previous value (if applicable). The user can right-click within the Variable Window and choose one of the options (i.e. edit or restore).

In some cases, there may be a lot of variables currently shown in the Variable Window. It is not convenient for the user to find a certain variable. Therefore, the Variable Window provides the filter operation for users. There are three filtering options:

filter by name – only provide a view of variables with the specified variable name

filter by value – only provide a view of variables with the specified value

filter by x-frame path – only provide a view of variables which are defined in the specified x-frame.



Variable	Value	X-Frame Path	Modified	<< Value History	Inserted
VD	7	D:/xvcl/src/xvcl/Test2/C....	original	7	no
VC	5	D:/xvcl/src/xvcl/Test2/C....	original	5	no
VB	[6,8,9,10]	D:/xvcl/src/xvcl/Test2/C....	original	[6,8,9,10]	no
VA	[6,8,9,10]	D:/xvcl/src/xvcl/Test2/C....	original	[6,8,9,10]	no

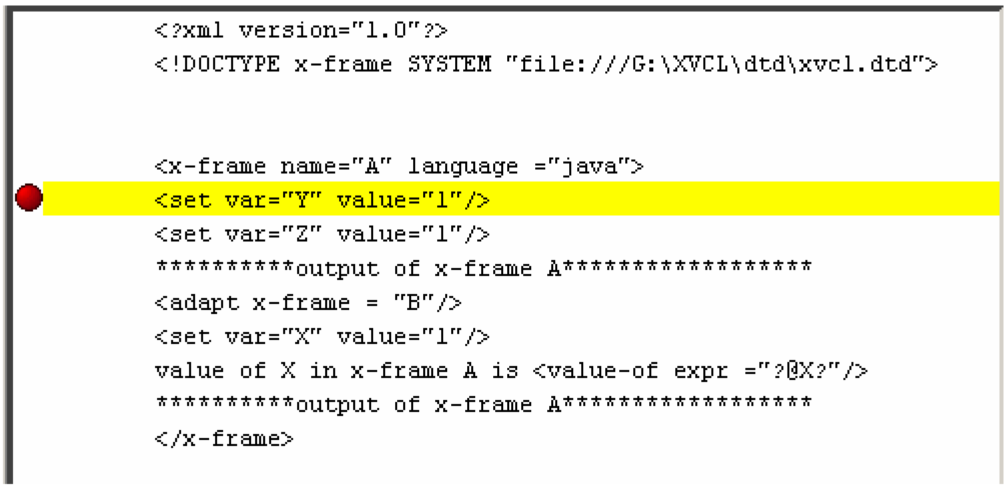
Filter Type: Variable name

Filter Value: VD

Figure 5-5 the Variable Window

5.4.4 Source Window

The Source Window is used to display the currently processing x-frame. In addition to displaying the frame contents, the view also indicates some other information. For instance, the current line being processed is highlighted in yellow.



```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file:///G:\XVCL\dtd\xvcl.dtd">


<x-frame name="A" language ="java">
<set var="Y" value="1"/>
<set var="Z" value="1"/>
**********output of x-frame A******************
<adapt x-frame = "B"/>
<set var="X" value="1"/>
value of X in x-frame A is <value-of expr ="?@X?"/>
**********output of x-frame A******************
</x-frame>
```

Figure 5-6 the Source Window

5.4.5 Output Window

In the Output Window, the user can view two types of outputs from IXP:

- Debugger Information

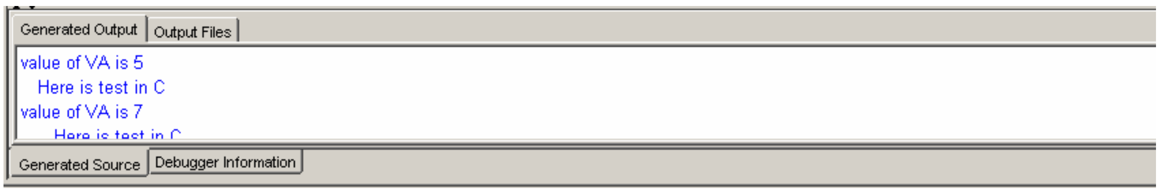
The *Debugger Information* tab shows the set of error messages emitted by IXP. The error information consists of name of the frame, number of the line at which the error occurred, and the error message text.

When the user modifies the value of a variable, the information is also traced and recorded into this tab.

When the processing terminates, the user can choose to save those information.

- Generated Source

The *Generated Source* tab shows the partial results that have been emitted by the processor. This lets the user see exactly how the current line being processed contributes to the output file. The user can also define the formats of the outputs.



```
value of VA is 5
   Here is test in C
value of VA is 7
      Here is test in C
```

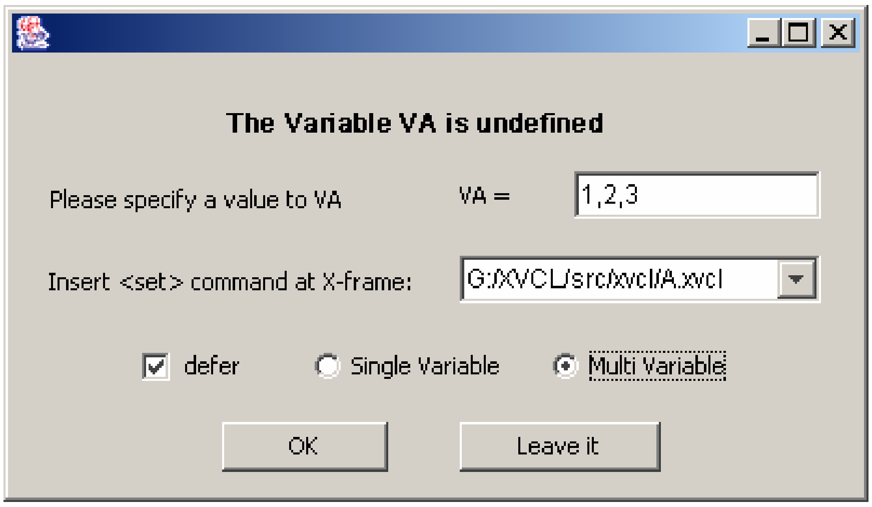
Figure 5-7 the Output Window

5.5 Detailed functionalities of the interactive XVCL processor

In this section, features of the interactive XVCL processor are described.

5.5.1 Setting a value to an undefined variable

IXP suspends the processing when a reference to an undefined variable is encountered. IXP pops a window (Figure 5-8) and asks the user to set a value to this variable.



The Variable VA is undefined

Please specify a value to VA   VA = 1,2,3

Insert <set> command at X-frame: G:/XVCL/src/xvcl/A.xvcl

[✓] defer   ○ Single Variable   ● Multi Variable

Figure 5-8 feeding a value to undefined variable VA

With the help of this window, the user can do the followings:

- provide a value for the undefined variable

  The type of a value is STRING, which is a mixture of any characters but “?”, “@” and “,”.

- specify in which x-frame this variable is to be defined

  The user is free to choose in which x-frame he wants to define this newly created variable. The candidate x-frames are those frames which are in the current adapting chain.

- specify the type of this variable

  The user can determine whether this variable is a single-value (e.g. 5) variable, or a multi-value (e.g. a list of values, “1, 2, 3”) variable. Of course, the user can choose only one type, not both at one time.

- assign defer-value for this variable

  The user can determine the defer-evaluation value of this variable. If the user decides defer-evaluation value is “yes”, the value of this variable is evaluated at the reference point rather than at the point where the variable is <set>.

In fact, what the user does here is equivalent to inserting a <set> or <set-multi> XVCL command into the user specified x-frame. For example, the user specifies an undefined variable “VA” as shown in Figure 5-8. It is equivalent to inserting the XVCL command

<set-multi var = “VA” value = “1, 2, 3” defer-evaluation = “yes”> into the x-frame “G:/XVCL/src/xvcl/A.xvcl”.

If the user fails to feed a value to an undefined variable, IXP will terminate processing of x-frameworks abnormally as the batch XVCL processor does.

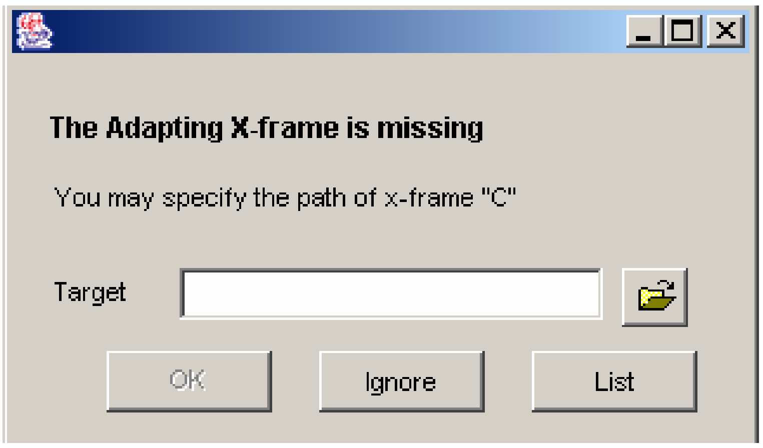
5.5.2 Indicating an x-frame to be adapted

IXP suspends the processing when <adapt> refers to a non-existing x-frame.

At this suspending point, the user can do the followings:

- Indicate the x-frame to be adapted at this processing point

  IXP pops a window (Figure 5-9) and asks the user to indicate the path to this x-frame.



The Adapting X-frame is missing

You may specify the path of x-frame "C"

Target

Figure 5-9 indicating the path for undefined x-frame C

  The user may browse the file system and designate a file as the undefined x-frame. IXP resumes the processing normally. The <adapt> x-command will instruct IXP to process the adapted x-frame which the user specified.

  If the user fails to find such an x-frame, he may ignore this undefined x-frame. IXP will resume processing and ignore this <adapt> XVCL command in this case.

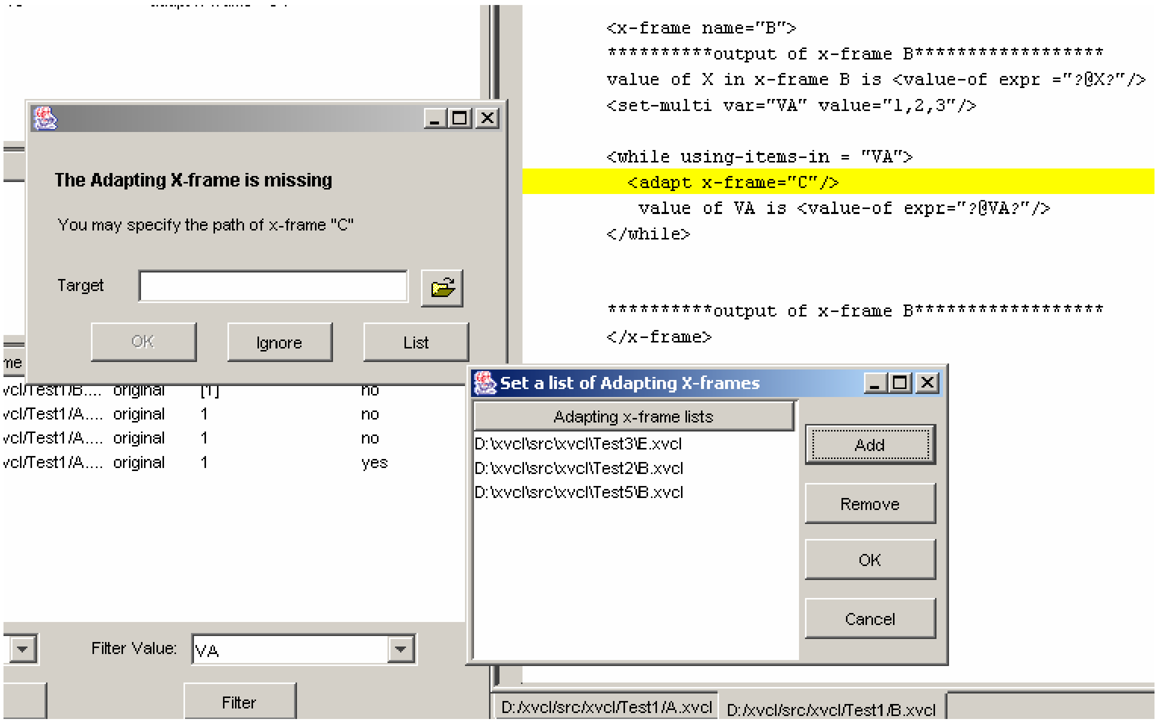
- Indicate an x-frame (or a list of x-frames) to be adapted at this processing point and at any points in subsequent processing

When a reference to an undefined adapting x-frame is encountered, the user may specify a list of x-frames at this processing point. These x-frames are adapted at current processing point and at any points where adapted x-frame is missing during the subsequent processing. Probably, the user chooses this option due to possible multiple missing adapted x-frames; especially in the case that the mistaken <adapt> command is inside a <while> loop. The following codes show such an example that the user is better to provide a list of x-frames at the current processing point (here, x-frame C.xvcl does not exist).

```
<while using-items-in = "VA">
    <adapt x-frame="C"/>
      value of VA is <value-of expr="?@VA?"/>
</while>
```

In the above example, the user, probably, may want to choose different adapted x-frames in different iterations of this <while> loop (Figure5-10).

An undefined x-frame “C” is encountered in the first iteration of this while loop. IXP pops the window and asks the user to indicate the path for x-frame “C”. The user can click “List” button. Another window pops up and it prompts the user to specify a list of x-frames for the current processing point and any points in the subsequent processing. As illustrated in Figure 5-10, the user specifies a list of x-frames at current processing point. They are “D:\xvcl\src\xvcl\Test3\E.xvcl”, “D:\xvcl\src\xvcl\Test2\B.xvcl”, and “D:\xvcl\src\xvcl\Test5\B.xvcl”. When processing resumes, the <adapt> x-command will instruct IXP to process the x-frame “D:\xvcl\src\xvcl\Test3\E.xvcl” in the first iteration of this <while> loop, “D:\xvcl\src\xvcl\Test2\B.xvcl” in the second iteration, and “D:\xvcl\src\xvcl\Test5\B.xvcl” in the third iteration.



```
<x-frame name="B">
**********output of x-frame B******************
value of X in x-frame B is <value-of expr ="?@X?"/>
<set-multi var="VA" value="1,2,3"/>

<while using-items-in = "VA">
  <adapt x-frame="C"/>
   value of VA is <value-of expr="?@VA?"/>
</while>


**********output of x-frame B******************
</x-frame>
```

The Adapting X-frame is missing

You may specify the path of x-frame "C"

Target

```
vcl/Test1/B....  original  [1]  no
vcl/Test1/A....  original  1    no
vcl/Test1/A....  original  1    no
vcl/Test1/A....  original  1    yes
```

Set a list of Adapting X-frames

Adapting x-frame lists

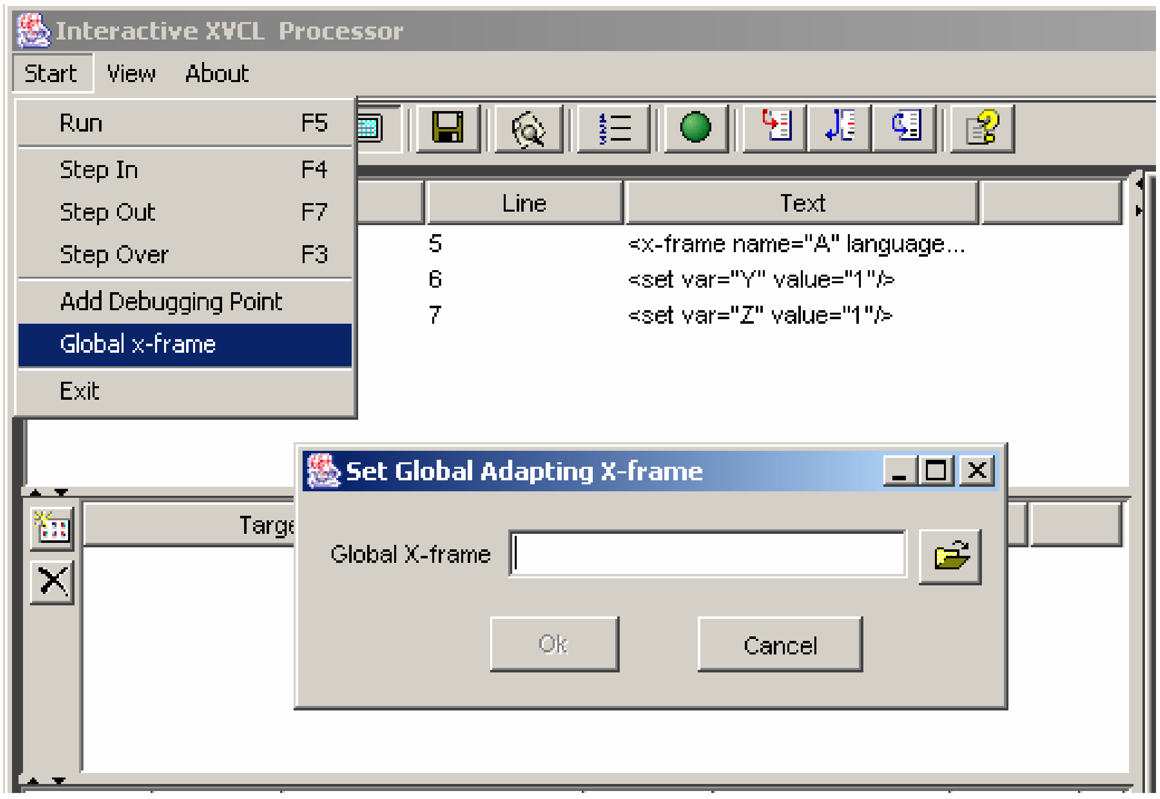
```
D:\xvcl\src\xvcl\Test3\E.xvcl
D:\xvcl\src\xvcl\Test2\B.xvcl
D:\xvcl\src\xvcl\Test5\B.xvcl
```

Filter Value: VA

Figure 5-10 providing a list of adapting x-frames

- specify up front a default action for any case of encountering an <adapt> of an undefined x-frame, in the whole processing

An attempt to adapt a non-existing x-frame is an error. However, such a situation often happens in x-framework under development. The user may specify a default x-frame to be used in place of any undefined x-frame.



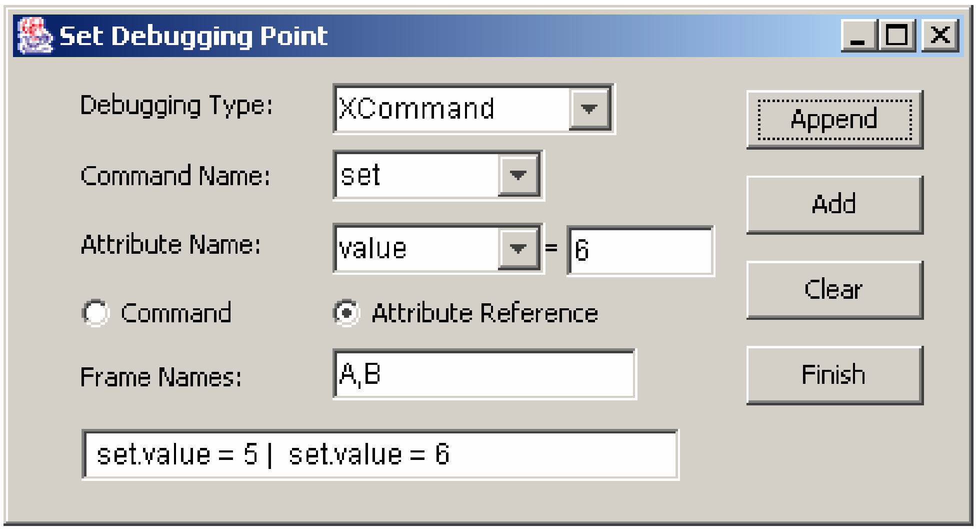
```
5 <x-frame name="A" language...
6 <set var="Y" value="1"/>
7 <set var="Z" value="1"/>
```

Figure 5-11 setting the global adapted x-frame

5.5.3 Setting debugging points

The user can set debugging points whenever the processing suspends. A debugging point is an XVCL command at which the user wants the processing of x-frameworks to suspend.

Once the processing suspends, the user can define a list of debugging points (see Figure 5-12). In order to specify a debugging point in a formal way, a language named Debugging Points Specification Language (DPSL) is introduced. All debugging points must be defined following the syntax and semantics of DPSL. The reader can get the complete specification of DPSL in the section 6.3.2 Debugging Points Specification Language (DPSL).


Set Debugging Point

Debugging Type: XCommand

Command Name: set

Attribute Name: value = 6

Command [✓ Attribute Reference]

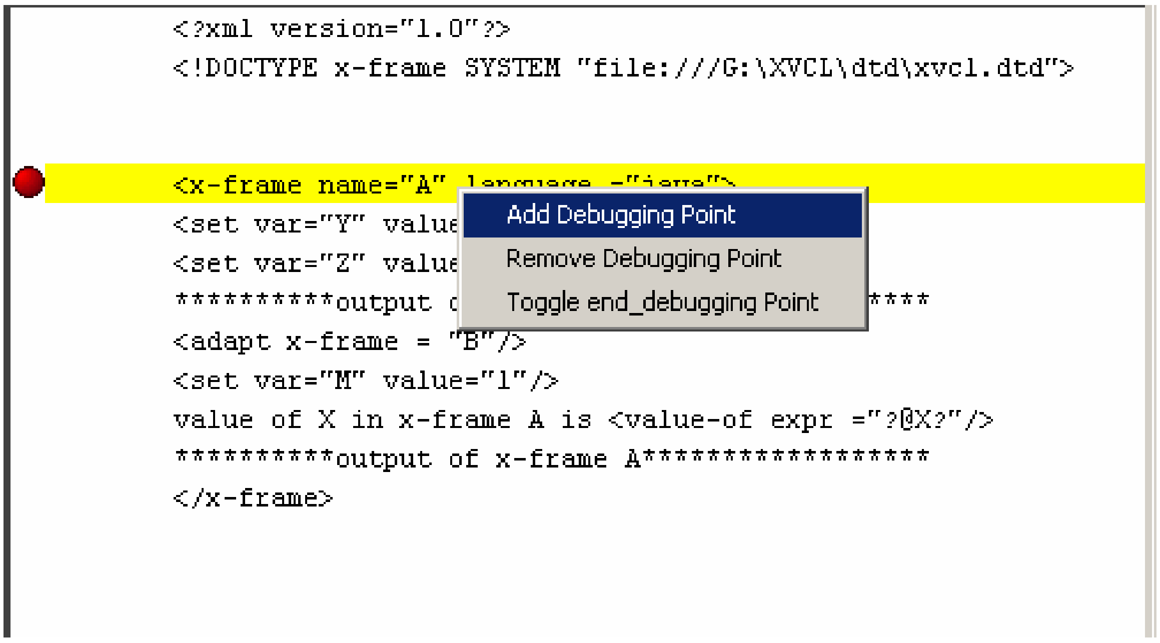
Frame Names: A,B

set.value = 5 | set.value = 6

Figure 5-12 Setting Debugging Point Dialog

With the help of “Setting Debugging Point Dialog”, the user can provide the complete specification of a debugging point. IXP keeps this debugging point and suspends the processing whenever it encounters an XVCL command at which this debugging point is set.

IXP provides an alternative way for the user to set a debugging point with the line type. The user can select a line and right-click within the Source Window (describe Source Window and other windows in the section 5.4). A context menu with three options pops up. Then the user can choose the “Add Debugging Point” option. This option allows the user to add a debugging point to the currently selected line. When a debugging point is confirmed to be set, a red cycle will show on the left margin of the Source Window.

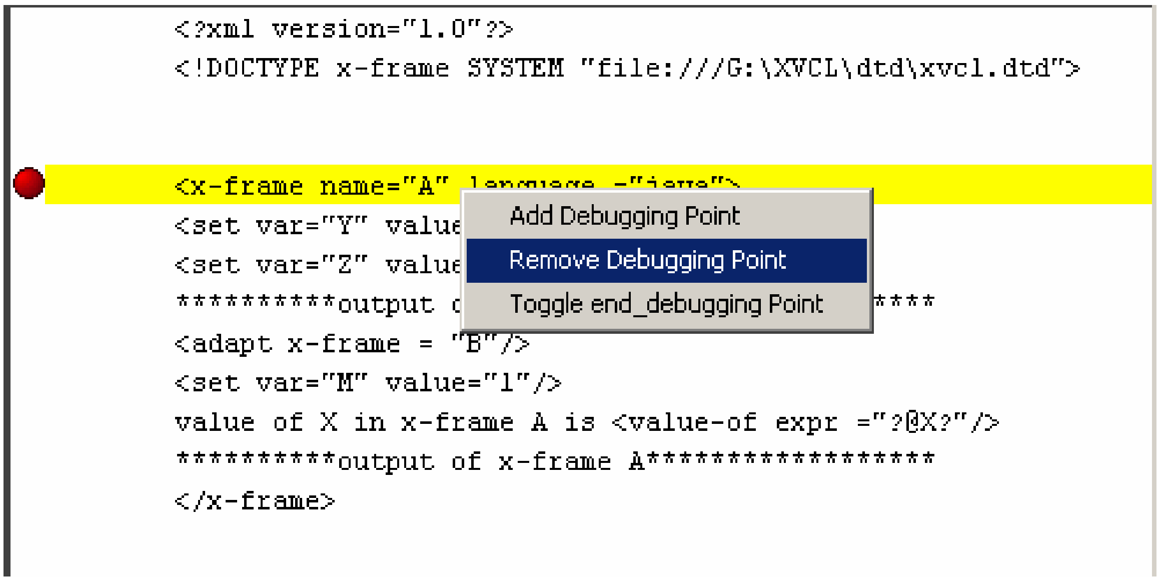


```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file:///G:\XVCL\dtd\xvcl.dtd">


<x-frame name="A" language ="java">
<set var="Y" value
<set var="Z" value
**********output c                    ****
<adapt x-frame = "B"/>
<set var="M" value="1"/>
value of X in x-frame A is <value-of expr ="?@X?"/>
**********output of x-frame A******************
</x-frame>
```

Figure 5-13 a red circle on the left margin of the window indicates a debugging point

5.5.4 Removing a debugging point

The user may remove a debugging point. Inside the Source Window, the user may right-click the mouse. A pop up menu will be shown. The user chooses the 2nd option “Remove Debugging Point” to remove a debugging point. The red cycle on the left margin of the Source Window will disappear if this debugging point is confirmed to be removed.



```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file:///G:\XVCL\dtd\xvcl.dtd">


<x-frame name="A" language ="java">
<set var="Y" value
<set var="Z" value
**********output c                    ****
<adapt x-frame = "B"/>
<set var="M" value="1"/>
value of X in x-frame A is <value-of expr ="?@X?"/>
**********output of x-frame A******************
</x-frame>
```

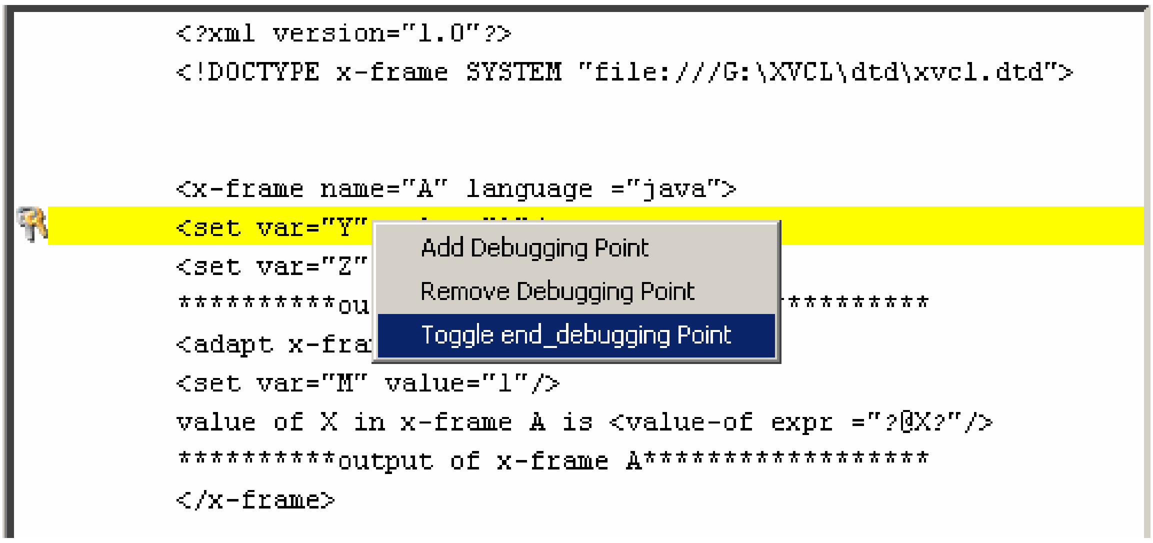
Figure 5-14 removing a debugging point

5.5.5 Setting end_debugging points

When an end_debugging point is encountered during the processing, all the debugging points in subsequent processing should be ignored.

Inside the Source Window, the user may right-click the mouse. A pop up menu will be shown. The user chooses the third option “Toggle end_debugging Point” to toggle an end_debugging point.

On the left margin of the Source Window, a “key” is used to indicate an end_debugging point.



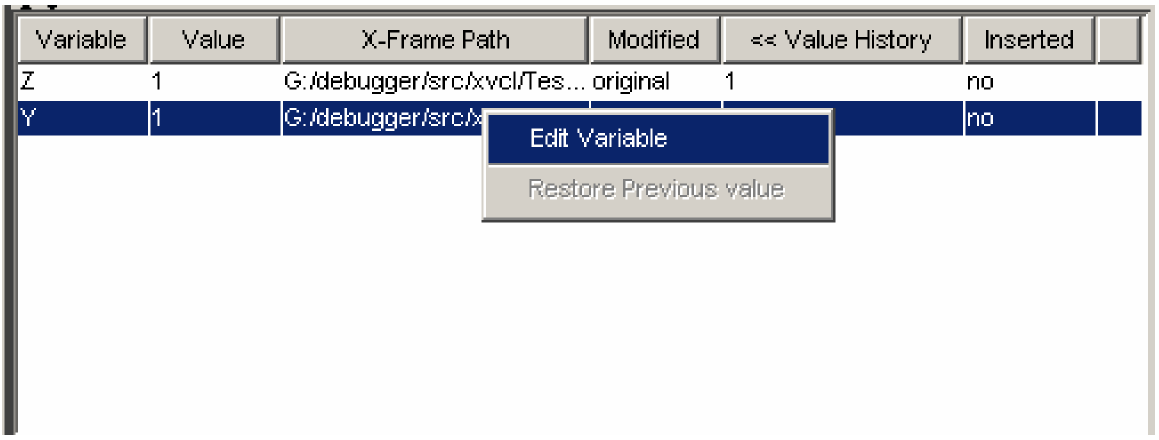
```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file:///G:\XVCL\dtd\xvcl.dtd">


<x-frame name="A" language ="java">
<set var="Y"
<set var="Z"
**********ou                                  *********
<adapt x-fra
<set var="M" value="1"/>
value of X in x-frame A is <value-of expr ="?@X?"/>
**********output of x-frame A******************
</x-frame>
```

Figure 5-15 toggling an end_debugging point

5.5.6 Modifying values of variables when the processing suspends

IXP makes it possible to alter the value of a given variable when processing suspends and have that change take effect immediately within the processing.

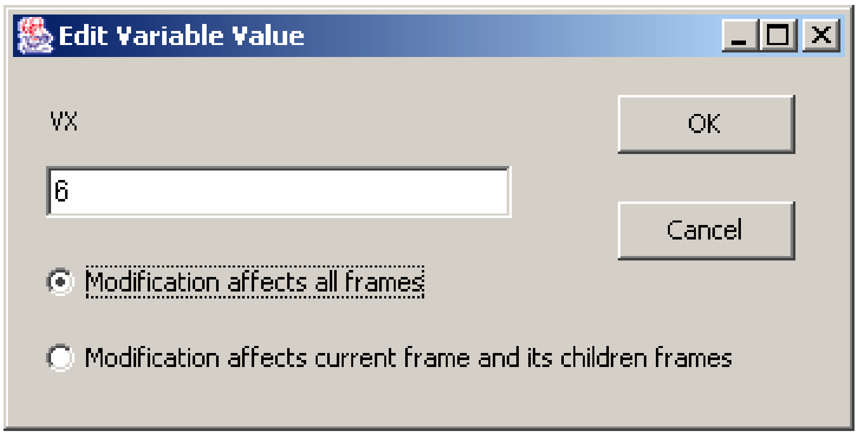
Inside the Variable Window, the user may select a variable and right click the mouse. A pop up menu shows. The user can choose the first option “Edit Variable” to modify the value of the selected variable.



Variable	Value	X-Frame Path	Modified	<< Value History	Inserted
Z	1	G:/debugger/src/xvcl/Tes...	original	1	no
Y	1	G:/debugger/src/x			no

Figure 5-16 editing Variable inside the Variable Window

The "Edit Variable Value" dialog then appears.



Edit Variable Value

VX

6

[✓ Modification affects all frames]

Modification affects current frame and its children frames

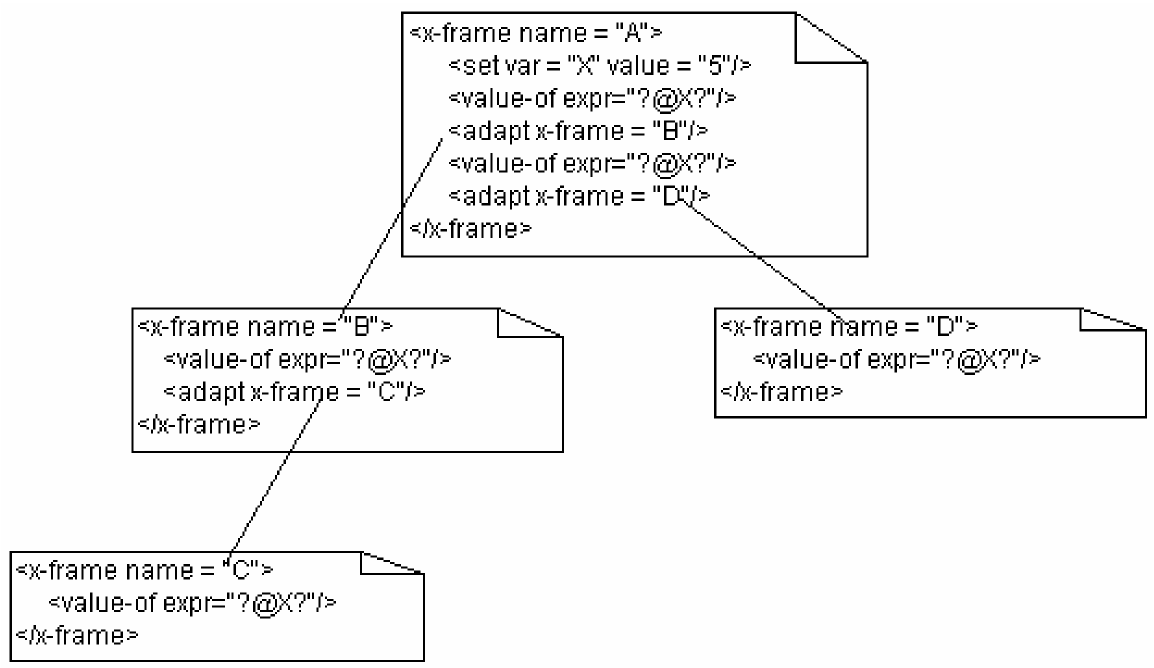
Figure 5-17 Edit Variable Dialog

The dialog displays the name of the variable that is being edited, as well as its current value. The user may also determine the lifetime scope of the new value. The dialog provides the following options for the user:

1. Use new value in all subsequently proceeded x-frames, both descendant from the x-frame containing the debugging point at which the value was modified and not descendant.

2. Use new value ONLY in descendent x-frames and old value of the variable in subsequently processed x-frames but not descendant from the x-frame containing the debugging point at which the value was modified.

We use an example to illustrate the ideas. The x-frame hierarchy is shown below.



```
<x-frame name = "A">
    <set var = "X" value = "5"/>
    <value-of expr="?@X?"/>
    <adapt x-frame = "B"/>
    <value-of expr="?@X?"/>
    <adapt x-frame = "D"/>
</x-frame>
```

```
<x-frame name = "B">
   <value-of expr="?@X?"/>
   <adapt x-frame = "C"/>
</x-frame>
```

```
<x-frame name = "D">
   <value-of expr="?@X?"/>
</x-frame>
```

```
<x-frame name = "C">
   <value-of expr="?@X?"/>
</x-frame>
```

Figure 5-18 an example of x-frame hierarchy

Consider the following scenario. IXP interrupts processing at XVCL command <x-frame name = “B”> in x-frame B. The user modifies value of the variable “X” to 100, and decides the scope of the modification is “Modification affects all frames which contain the reference to this variable” (option 1). After processing completes, the result emitted is:

```
X = 5 (in x-frame A, before adapting B)

X = 100 (in x-frame B)

X = 100 (in x-frame C)

X = 100 (in x-frame A, after adapting B)
```

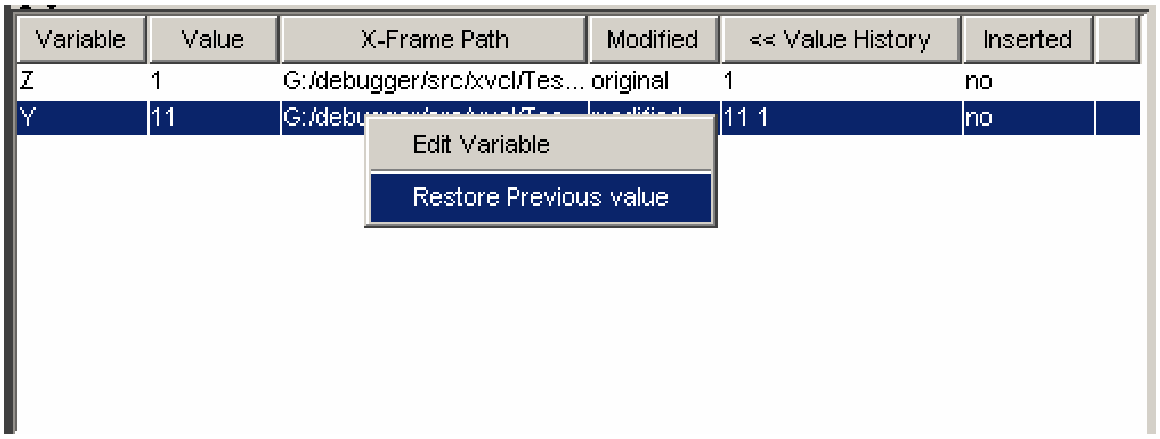
However, if the user decides the scope of the modification is “Modification affects current frame and its children frames ONLY” (option 2) when processing suspends at <x-frame name =”B”>. Instead, after processing completes, the result is:

```
X = 5 (in x-frame A, before adapting B)
X = 100 (in x-frame B)
X = 100 (in x-frame C)
X = 5 (in x-frame A, after adapting B)
```

Modifying values of variables when IXP is paused is quite a useful feature for debugging purpose. The user can use this function to trace the processing sequence in order to find the bug.

In some situations, the user may not be satisfied with the new value. He wants to discard the change. In this case, the user is able to restore the value of this variable to its previous value.

Inside the Variable Window, the user may select a variable and right click the mouse. A pop up menu shows. The user can choose the second option “Restore Previous value” to restore the previous value.



Variable	Value	X-Frame Path	Modified	<< Value History	Inserted
Z	1	G:/debugger/src/xvcl/Tes...	original	1	no
Y	11	G:/debu[illegible]	[illegible]	11 1	no

Figure 5-19 restoring the value of a variable

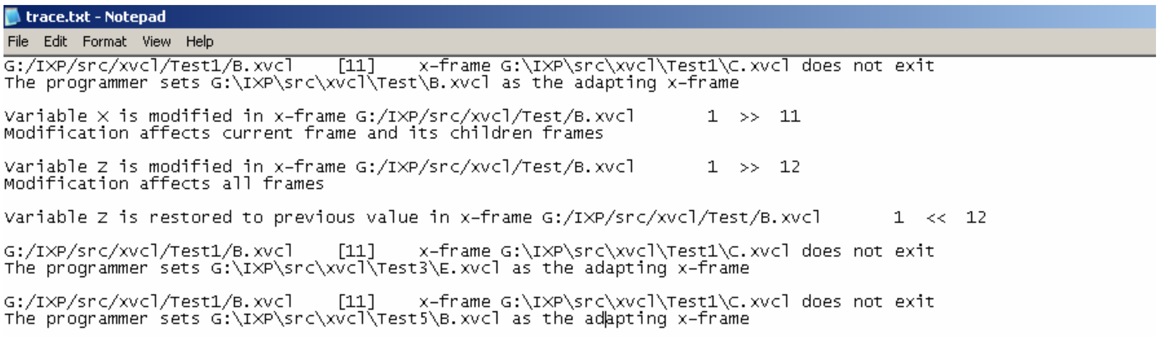
5.5.7 Storing the trace information

The trace information of the debugging is saved in a file named "trace.txt" in the working directory. The trace information includes:

- the information of an undefined variable: its name and location (in which line of which x-frame)
- the user's action when IXP encounters a reference to an undefined variable
- the information of an undefined x-frame: its name and location
- the user's action when IXP encounters a reference to an undefined x-frame
- the information of the variable modification: the old and new value of the variable, and the scope of the modification
- the information of the variable restoration

The user may examine the output generated at the end of processing. If he thinks some modifications are relevant, he may update x-frameworks accordingly. Otherwise, he may just ignore them.

The Figure 5-20 shows a sample of “trace.txt”.



```
G:/IXP/src/xvcl/Test1/B.xvcl     [11]     x-frame G:\IXP\src\xvcl\Test1\C.xvcl does not exit
The programmer sets G:\IXP\src\xvcl\Test\B.xvcl as the adapting x-frame

Variable X is modified in x-frame G:/IXP/src/xvcl/Test/B.xvcl       1  >>  11
Modification affects current frame and its children frames

Variable Z is modified in x-frame G:/IXP/src/xvcl/Test/B.xvcl       1  >>  12
Modification affects all frames

Variable Z is restored to previous value in x-frame G:/IXP/src/xvcl/Test/B.xvcl       1  <<  12

G:/IXP/src/xvcl/Test1/B.xvcl     [11]     x-frame G:\IXP\src\xvcl\Test1\C.xvcl does not exit
The programmer sets G:\IXP\src\xvcl\Test3\E.xvcl as the adapting x-frame

G:/IXP/src/xvcl/Test1/B.xvcl     [11]     x-frame G:\IXP\src\xvcl\Test1\C.xvcl does not exit
The programmer sets G:\IXP\src\xvcl\Test5\B.xvcl as the adapting x-frame
```

Figure 5-20 a sample of trace.txt

5.5.8 Single-stepping

There are four different step functions: step through, step out, step over and skip over. The user can switch between stepping mode at any time during the processing, which would affect subsequent stepping.

- Step Through

In Step Through mode, IXP executes XVCL commands one-by-one, stopping at each command. Whenever a <set> command is executed, a variable along with the assigned value (and some other information) is shown in the Variable Window. Execution of <adapt> command leads the user to the adapted x-frame.

In IXP, Step Through is bound with the shortcut key F4.

The following shows a simple example.

x-frame A	x-frame B
1. <x-frame name="A">	1. <x-frame name="B">
2. <set var="X" value="1"/>	2. <value-of expr ="?@X?"/>

```
3.  <adapt x-frame = "B"/>          3.  </x-frame>
4.  <set var="Y" value="1"/>
5.  </x-frame>
```

If the user click F4 at line 3 of x-frame A, IXP advances to line 1 of x-frame B and processes commands in x-frame B.

- Step Out

Step Out brings the user to the first command to be processed after the XVCL structure (includes <x-frame>, <while> and <select>) directly containing the current command. For example, if the user is at command directly contained in the <while> structure - the user will Step Out to the first command to be processed after <while>. Stepping Out a command directly contained in an x-frame A will lead the user to the first command after <adapt A>.

Notice that all the commands in-between the current command and command after the structure are executed, but IXP does not stop at them.

- Step over

Step Over allows the user to step through the current x-frame line-by-line while ignoring any adapted x-frames. In the example above, if the execution stops at line 3 of x-frame A and the user clicks Step Over, IXP moves directly to line 4 of x-frame A, and the adapted x-frame B is ignored although it is still executed.

- Skip over

Skip Over is almost the same as Step Over. The only difference is that Step Over does process adapted x-frames, while Skip Over does not.

5.5.9 Setting outputs formatting

The user can set the format of the outputs.

To prevent the Output Window from being flooded by thousands of output lines, the user may set the maximum number of lines to be displayed in that window. If the user sets “maximum number of output lines” to 500, and the number of output lines reaches 500, the Output Window will refresh the window, and keep only most recently produced N output lines (N is defined by “minimum number of output lines”). Other parts of outputs will be truncated.

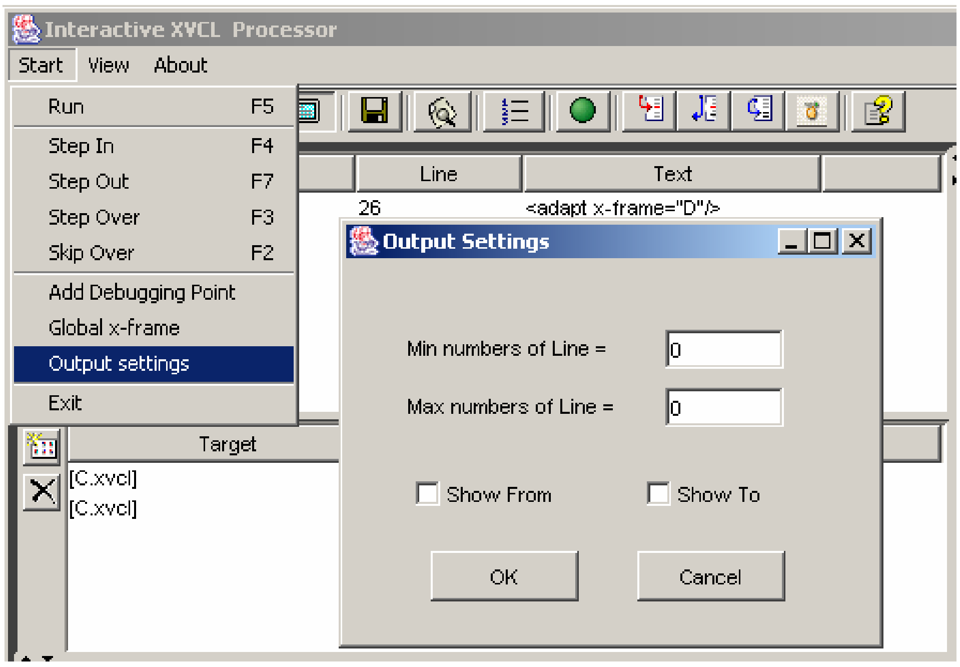
Notes:

1. If the user set “maximum number of output lines” to 0, the Output Window will keep all the output generated.

2. “Minimum number of output lines” must be less than or equal to “maximum number of output lines”.

If the checkbox “Show From” is ticked, the Output Window will show which parts of the output comes from which x-frame.

If the checkbox “Show To” is ticked, the Output Window will show which parts of the output goes to which output file.



```
26        <adapt x-frame="D"/>
[C.xvcl]
[C.xvcl]
```

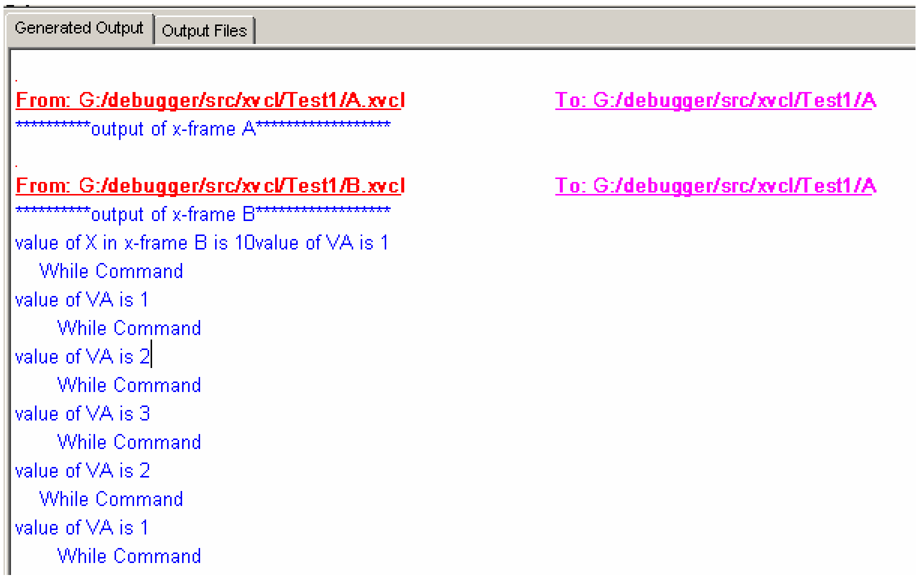
Min numbers of Line = 0

Max numbers of Line = 0

[ ] Show From [ ] Show To

Figure 5-21 output setting options

The following figure shows what Output Window looks like if the option “Show From” and “Show To” are set.



```
From: G:/debugger/src/xvcl/Test1/A.xvcl            To: G:/debugger/src/xvcl/Test1/A
**********output of x-frame A******************

From: G:/debugger/src/xvcl/Test1/B.xvcl            To: G:/debugger/src/xvcl/Test1/A
**********output of x-frame B******************
value of X in x-frame B is 10value of VA is 1
   While Command
value of VA is 1
     While Command
value of VA is 2
     While Command
value of VA is 3
     While Command
value of VA is 2
   While Command
value of VA is 1
     While Command
```

Figure 5-22 showing outputs with options “Show From” and “Show To”

Chapter 6

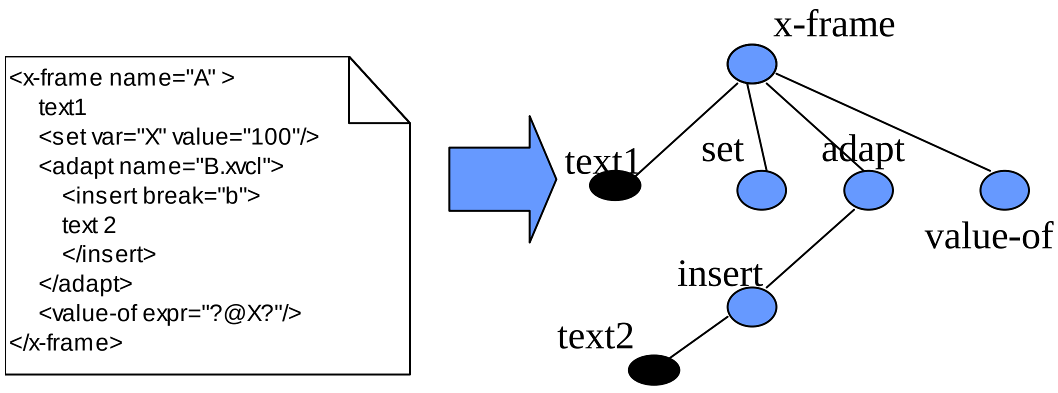
Design and implementation of the interactive XVCL processor

6.1 Design of the batch XVCL processor

IXP was developed based on the batch XVCL processor. In this part, we briefly describe the implementation of the batch XVCL processor.

6.1.1 JDOM based XVCL processor

X-frames are XML files which contain XVCL commands, and Textual Contents between end and start tags [11]. In fact, an x-frame can be represented as a tree. The following figure shows a tree representation of an x-frame.



```
<x-frame name="A" >
   text1
   <set var="X" value="100"/>
   <adapt name="B.xvcl">
      <insert break="b">
      text 2
      </insert>
   </adapt>
   <value-of expr="?@X?"/>
</x-frame>
```

Figure 6-1 tree view of an x-frame

To translate an x-frame to its tree representation, we need a tool support. In our XVCL implementation, we use JDOM, which is a Java-based "document object model" for XML files [8]. It's an alternative to DOM and SAX, although it integrates well with both DOM and SAX. Like DOM, JDOM represents an XML document as a tree composed of elements, attributes, comments, processing instructions, text nodes, CDATA sections, and so forth. JDOM is not itself a parser. Instead it depends on a SAX parser with a custom `ContentHandler` to parse XML documents and build JDOM models from them. With JDOM, we can do the following:

- represent an x-frame as a tree

  JDOM uses a factory (XVCLBuilderFactory, in our implementation) to create the object of each node in the tree.

- manipulate the tree, such as navigating (e.g. getParent(), getChildren(),etc), adding (e.g. addContent()) and removing (e.g. remove(content))

- manipulate the nodes, such getting attribute, getting attribute's value, getting text

6.1.2 Design concepts of the batch XVCL processor

The batch XVCL processor interprets each XVCL command in x-frames and gathers the emitted result of each XVCL command as output. To make JDOM tree parse itself, we extend it to XVCL command tree. We need create a class for each XVCL command and create a JDOM factory for returning XVCL command class's objects as JDOM Tree Nodes [8]. Figure 6-2 shows class diagram of XVCL command classes.

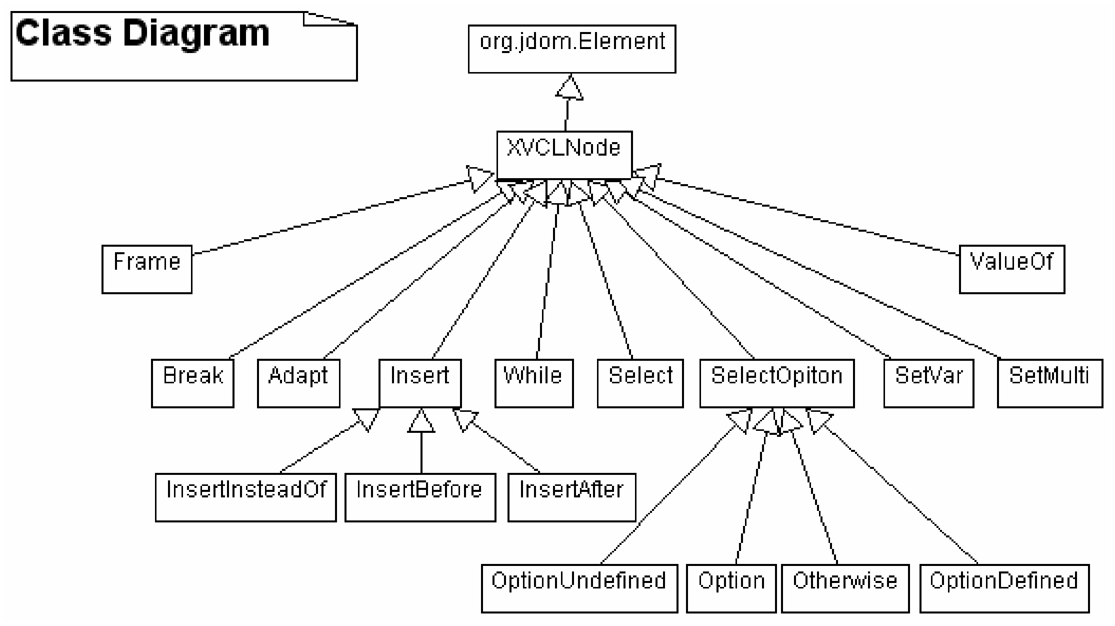


Figure 6-2 class diagram of XVCL commands

For each XVCL command class, we create three methods:

- *void parseAttributes()*: interprets attributes and values of each command
- *StringBuffer emit()*: writes the result (including this command's output and its children's) to file or return it to it's parent.
- *void parse()*: parses all children nodes and gather outputs from children one by one.

XVCLNode is a superclass for all XVCL commands classes. The above three methods are defined in XVCLNode class. However, each XVCL command class may override them because one may have different way to parse its attributes, emit results and parse its children nodes from another.

6.2 Design of debugger functionalities on top of the batch XVCL processor

Based on the general requirements for XVCL analysis and debugging, we designed a suitable debugging tool on the batch XVCL processor.

Firstly, as we mentioned, the XVCL processor is in the batch mode, since it processes an x-framework without stopping. However, IXP allows the user to set a debugging point at which the processing suspends or to single-step through the XVCL program. As a result, IXP can stop anywhere during the processing. Reflecting on the design aspect, the interactive processor is a runnable object with multiple threads, comparing with the batch XVCL processor, which is running with a single thread.

Secondly, IXP allows the user to modify variable values during the processing. To trace the possible variable modifications and restorations, we need keep more information about the variables. The original design of the symbol table is far from sufficient. As a result, we should expand the symbol table to hold more information, such as variable values history, variable modification flag and variable insertion flag.

Thirdly, to set debugging points in a formal way, we need define a language to formulate the syntax and semantic of debugging points specification. DPSL is designed for this purpose. Details are given later.

Fourthly, as we introduced the concepts like debugging point and output lines, the new data objects, as well as their interfaces are designed. For example, DP is an object to hold a debugging point and DPT is an object to hold all debugging points currently set by the user.

Their interface operations are defined on the top of the batch XVCL processor.

In summary, IXP imposes the debugging functionalities on the batch XVCL processor in the following ways:

- includes a modified symbol table that can be viewed and changed during the processing
- includes the specification of the debugging points via DPSL (Debugging Points Specification Language)
- includes a debugging point table that holds all debugging points that are currently set
- includes four stepping operations: step through, step out, step over, and skip over
- includes outputs formatting: such as show only recent N lines of outputs in the Output Window
- includes some other modifications made to the implementation of the batch XVCL processor

6.3 Implementation of IXP

6.3.1 The symbol table

6.3.1.1 The symbol table in the batch XVCL processor

XVCL variables provide means for controlling the x-framework customization process. The batch XVCL processor uses symbol tables to store all variables defined so far with current values assigned to them. Each x-frame has its own partial symbol table, keeping variables defined in this x-frame. Partial symbol tables for all x-frames form the symbol table for the x-framework. Figure 6-3 shows an example of a symbol table.

Name	*Value*
X	5
Y	Z
Z	V
Multi	1,2,3

Figure 6-3 an example of the symbol table

6.3.1.2 Variable scoping rules

To explain the symbol table, we must understand *variable scoping rules* first. The rules are the same for both single-value and multi-value variables. The <set> (or <set-multi>) commands in the ancestor x-frame take precedence over <set> commands in its descendent x-frames. For example, once x-frame A 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 A can reset the value of variable v. Variable v becomes undefined as soon as the processor returns the processing to the parent x-frame that adapts x-frame A.

6.3.1.3 Extended symbol table in IXP

In IXP, we must extend the symbol table for the following reasons:

1. As we mentioned in Chapter 5, the user may modify values of variables when processing suspends. Both the new value and old values of a modified variable should be kept. Therefore only storing a variable’s name and value is insufficient. We extend the symbol table so that it also can tell whether a variable has been modified, and what its old values were. The extended symbol table now also contains a list of values that have been assigned to this variable including the current assigned value.

2. In addition, when IXP encounters a reference to an undefined variable, the user may define this variable. After that, IXP inserts this variable into the symbol table of the x-frame that is specified by the user. We extend the symbol table so that it can indicate whether a variable is newly inserted or not.

The format of the extended symbol table is shown in Figure 6-4:

Name	**Value**	**X-frame path**	**Modified**	**Value history**	**Inserted**

Figure 6-4 the format of the extended symbol table

Field definition:

Name – variable name

Value – the current value assigned

X-frame path – indicates in which x-frame this variable is defined

Modified – indicates whether the variable has been modified (has value “modified”) or not (has value “original”).

Value history – specifies a list keeping all values that have been assigned to this variable, including the current assigned value. The newest value is inserted to the head of the list.

Inserted – has two values (“yes” or “no”). “yes” indicates the variable is undefined before IXP encounters its reference, but the user feeds its value during the processing. “no” indicates otherwise.

6.3.1.4 Implementation of the symbol table in IXP

We use a data structure called “VariableInfo” to hold extra information of a variable such as “X-frame path”, “Modified”, “Value history” and “Inserted”. VariableInfo class extends

Variable class. It inherits all attribute members and interface operations of Variable class. Therefore, it also holds the basic information of a variable, such as "current value of the variable", "whether the variable is a multi value variable or a single value variable", and "whether the variable is defined to defer evaluation". In short, VariableInfo class is another representation of a variable, but it provides more information compared with Variable class.

```
class VariableInfo extends Variable{
        String x-framepath;
        List valuehistory;
        boolean modified;
        boolean inserted;
}
```

VariableInfo provides a set of interface operations for the user to set and retrieve necessary information of a variable. The complete API of VariableInfo class can be found in Appendix A.

A symbol table is a Hashtable implemented in XVCLNode Class, which is a superclass of all XVCL command classes. The structure of a symbol table looks like {Name, VariableInfo}.

Given a variable name, to get the corresponding variable, we use the following algorithm:

```
checking x-frame: = current processing x-frame;
boolean found := false;
while (checking x-frame ! = null && !found) {
        search through the symbol table of checking x-frame;
        if find a matched variable , found:= true, retrieve this variable
        else checking x-frame:= parent x-frame which adapts checking x-frame;
}
if checking x-frame == null, it means the variable is undefined.
```

To get the current value of a variable, we retrieve the head element of "valuehistory" (a field of VariableInfo class as we described above) of this variable.

Based on the *variable scoping rules*, when we are going to store a variable into a frame's symbol table, we need check whether this variable exists in the symbol tables of ancestor frames. If the variable is not defined in ancestor frames, it will be stored to the nearest symbol table. Otherwise, it will be ignored.

6.3.1.5 Implementation of the variable modification

In Chapter 5, we mentioned, when a user modifies the value of a variable, he has two options for the lifetime scope of the modification (Section 5.4.4).

1. Modification affects all frames which have a reference to this variable

2. Modification only affects the current frame and its children frame (if any)

If the user chooses option 2, the new value is inserted as the head element of "valuehistory" (See description in section 6.2.1.3) of this variable. However, when IXP returns the processing to the parent x-frame, the new value must be removed from "valuehistory". Thus, the modification won't affect frames which are not descendants of the x-frame where the variable was modified.

So we extend the "valuehistory" field of VaribleInfo class in our implementation. Now it is a list keeping all values that have been assigned to the variable, as well as names of x-frame at which the variable was modified. If the user modifies a variable with option 1, we consider the modification takes place at the x-frame where the variable is originally defined.

Consider the following scenario. X-frame A adapts x-frame B. Variable "X" is defined in x-frame A with value 11. Its value is changed to 5 in x-frame B. The modification only affects frame B and its children frame (option 2). Thus, the "valuehistory" field of variable "X" is [(5, B), (11, A)] (Notice, the newest value is inserted to the head of the list) in this stage.

With new specification of “valuehistory”, an example of symbol table looks like:

Name	*Value*	*X-frame path*	*Modified*	*Value history*	*Inserted*
X	**5**	**C:\A.xvcl**	**Modified**	**(5,B) (11,A)**	**no**
Y	**10**	**C:\A.xvcl**	**Original**	**(10,A)**	**yes**

Figure 6-5 the symbol table of x-frame A after X is modified in x-frame B (option 1)

When IXP returns the processing to x-frame A, (5, B) is removed from the “valuehistory” of “X”, since IXP has already processed x-frame B and B is a child frame of A. The symbol table of x-frame A now is:

Name	*Value*	*X-frame path*	*Modified*	*Value history*	*Inserted*
X	**11**	**C:\A.xvcl**	**Original**	**(11,A)**	**no**
Y	**10**	**C:\A.xvcl**	**Original**	**(10,A)**	**yes**

Figure 6-6 the Symbol table of x-frame A after IXP returns processing to x-frame A

However, in x-frame B, if the user modifies the value of X to 5, but with option 1. After this modification, the symbol table of x-frame A looks like:

Name	*Value*	*X-frame path*	*Modified*	*Value history*	*Inserted*
X	**5**	**C:\A.xvcl**	**Modified**	**(5,A) (11,A)**	**no**
Y	**10**	**C:\A.xvcl**	**Original**	**(10,A)**	**yes**

Figure 6-7 the Symbol table of x-frame A after X is modified in x-frame B (option 2)

When IXP returns processing to x-frame A, however, the symbol table won’t change. The modification still takes effect at x-frame A as the user expects.

6.3.2 Debugging Points Specification Language (DPSL)

IXP, a debugger for XVCL, provides a user interface for users to set debugging points when the processing of an x-framework suspends. A debugging point designates a location in user’s program where program temporarily stops executing. When a bug occurs in the user’s application, the user can set a debugging point to suspend processing just before the unexpected behavior occurs. Once processing suspends, the user can examine the variable values and partial results emitted to estimate the possible reasons of error; the user can also set more debugging points for future debugging purpose if necessary.

In our XVCL, a debugging point can only refer to an XVCL command, not Texture Content. To facilitate the user to specify a debugging point, we design a language, Debugging Points Specification Language (DPSL, for short).

6.3.2.1 Syntax of DPSL

The syntax of DPSL is defined as:

DPSL :: = **debugging-point** (‘,’ **debugging-point**)*

debugging-point :: = ‘[‘**frame**’]’.**linenum** | ‘[‘**frame-list**’]’.**xcommand**

xcommand ::= **xcommand** ‘|’ **xcommand –term**

| **xcommand-term**

xcommand-term ::= **xcommand-term** ‘&’ **xcommand-factor**

| **xcommand-factor**

xcommand-factor ::= **command-spec** | ‘(‘ **xcommand** ‘)’ | ‘!’ **xcommand-factor**

command-spec :: = **command-name** | **attribute-ref**

attribute-ref :: = **command-name** ’.’ **attribute-name** ‘=’ **value**

frame-list ::=   **frame**(‘,’**frame**)*

frame ::= an x-frame name

command-name ::= an XVCL command name

attribute-name ::= name of an attribute which belongs to an XVCL command

value :: = **STRING | INTEGER**

linenum :: = **INTEGER**

STRING ::= a mixture of any characters but “?”, “@” and “,”

INTEGER ::= a positive integer number

6.3.2.2 Language description

DPSL specifies a list of debugging points.

There are two different types of debugging points:

- Line (‘[‘frame’]’.linnum)

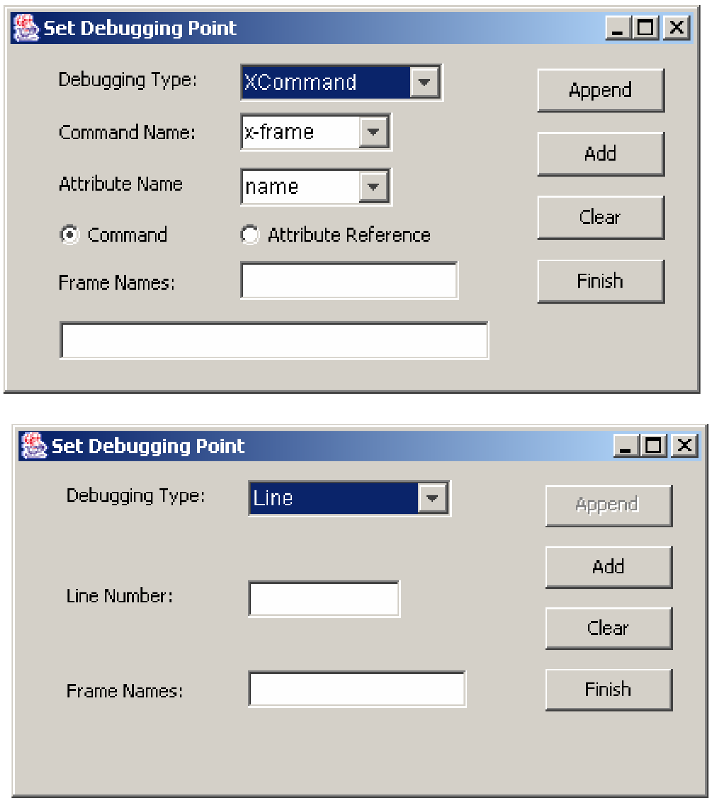
  A line type debugging point causes the IXP to halt when it reaches a specific line in a particular x-frame. In the case of line suspends, the *frame* is the x-frame in which the debugging point is set, and the *linnum* is the line number at which IXP will pause. An example of a debugging point with line type is ‘[A].100’. It means IXP should suspend processing of x-frameworks at the 100th line of x-frame A.

- XVCL command (‘[‘frame-list’]’.xcommand)

  An XVCL command debugging point causes IXP to halt whenever a specific XVCL command is encountered. In this case, the *frame-list* is the list of frames in which the debugging point is set, and *xcommand* is a description of the XVCL command at which

IXP will interrupt processing. How to specify an XVCL command in DPSL is explained later.

The user can decide the type of a debugging point and use different ways to represent it. IXP provides “Setting Debugging Point Dialog” for the user to specify a debugging point. The dialog will show different appearances based on the type of the debugging point.



Debugging Type: XCommand
Command Name: x-frame
Attribute Name name
[✓ Command] Attribute Reference
Frame Names:

Debugging Type: Line
Line Number:
Frame Names:

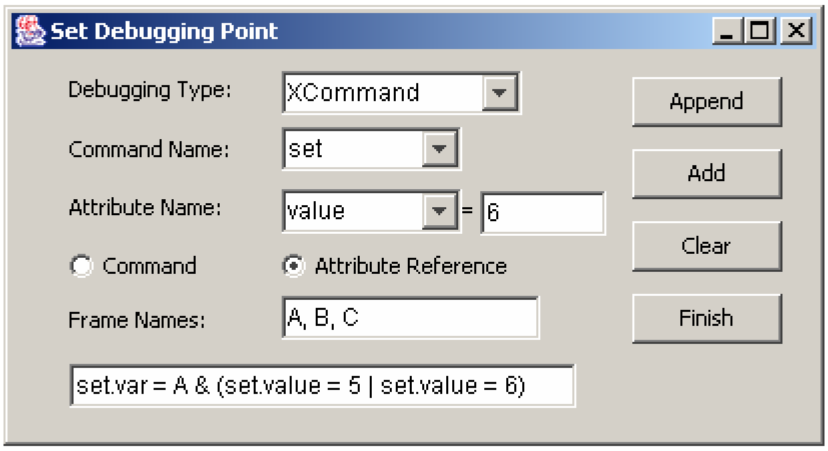
Figure 6-8 two views of Setting Debugging Point Dialog based on type of a debugging point

An XVCL command can be represented as an expression. The logical operators used in the expression include *and* (“&”), *or* (“|”) and *not* (“!”). Among them, *not* has the highest precedence, and *or* has the lowest precedence. We also use parentheses (“(“, “)”) to preserve the precedence. The literal of the expression is either an XVCL command name (e.g. set, refers to a <set> command) or an attribute reference of an XVCL command (e.g. set.var = “A”, refers to a <set> command whose attribute “var” has value “A”).

An example of such an expression is ‘set.var = “A” & (set.value = “5” | set.value = “6”)’.It specifies a <set> XVCL command, which assigns value “5” or “6” defined in *value* attribute to a single-value variable “A” defined in *var* attribute. (in fact, it is equivalent to <set var = “A” value = “5”> or <set var = “A” value = “6”>).

An example of a debugging point with XVCL command type is ‘[A, B, C]. set.var = “A” & (set.value = “5” | set.value = “6”)’. The *frame-list* [A, B, C] restricts this debugging point is set only in x-frames A, B and C.

The following figure shows how to set this debugging point with the help of “Setting Debugging Point Dialog”.


Set Debugging Point

Debugging Type: XCommand

Command Name: set

Attribute Name: value = 6

Command [✓ Attribute Reference]

Frame Names: A, B, C

set.var = A & (set.value = 5 | set.value = 6)

Figure 6-9 setting a debugging point with the help of the Dialog

DPSL also supports wildcard character. Wildcard character is a character that may be substituted for any of the defined subsets of all possible characters. DPSL supports asterisk (“*”). Asterisk usually substitutes for any one or more characters. For example, if we specify “s*” for an XVCL command, it can refer to any of <set>, <set-multi>, and <select> XVCL commands. A debugging point specification with wildcard characters, ‘[*].adapt.x-frame = A*’, means IXP suspends processing in *any* x-frame, whenever an <adapt> command that ‘adapts’ an x-frame whose name starts with ‘A’ (e.g. A, A1, AB, etc) is encountered.

6.3.2.3 Limitations of DPSL definition

There are some limitations in current DPSL definition.

1. The content of a line at which a debugging point with line type is set must be an XVCL command. If not (e.g. a line which is textual content or comment), IXP will ignore this debugging point.
2. As we mentioned in the previous section, in a debugging point with XVCL command type, the specification of an XVCL command is an expression. A literal of the expression is either an XVCL command name or an attribute reference of a command. In a single expression, XVCL commands that any literal involves must be in the same x-command type. Otherwise, it is an illegal specification. For example, set.value = “5” | set.var = “X” is a legal specification, but set.value = “5” & adapt.once = “yes” is illegal, since <set> and <adapt> are not in the same type. The reason to set this limitation is that a debugging point specification like ‘set.value = “5” & adapt.once = “yes”’ is meaningless, and non-interpretable.

6.3.2.4 Interpret DPSL

Once DPSL is defined, we have two steps to interpret the language.

- Normalization

This step is only for debugging points with XVCL command type. For example, ‘[A, B, C].set.var = “A” & (set.value = “5” | set.value = “6”)’ is an XVCL command type debugging point. It is a frame-list (i.e. [A, B, C]) followed by an expression. We need interpret the expression so that IXP can understand it. To simplify the interpretation, we transform the expression to its Disjunctive Norm Form (DNF) first if it is not in DNF. The expression in the example is not in DNF.

The DNF normalization is described in the following.

Step 1: we construct a tree view of the expression. The leaf nodes of the tree are literals of the expression, while internal nodes are logical operators. A tree view of the expression in the example is:

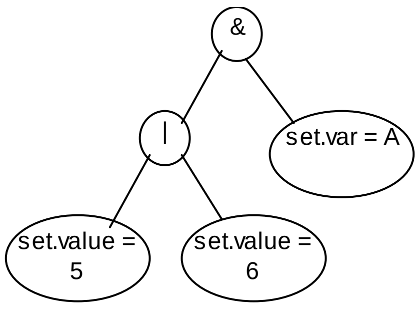


Figure 6-10 the tree view of the expression in the example

Step 2: we check whether an expression is in DNF or not by examining its corresponding tree structure. The rule is no “|” node is descendant of any “&” node in the tree. In our example, since the left child of ‘&’ node is a ‘|’ node, we can determine the corresponding expression is not in DNF.

In this step, if we get a negative answer (i.e. expression is not in DNF), we proceed to step 3. Otherwise, we terminate the normalization procedure, since the expression is in DNF already.

Step 3: we reconstruct the tree based on the following mathematical rules:

- ✓ (A|B) & C = A&C | B&C

- ✓ ! (A|B) = !A & !B
- ✓ ! (A&B) = !A | !B

In our example, after reconstruction, we get a new tree view which representing an expression in DNF.

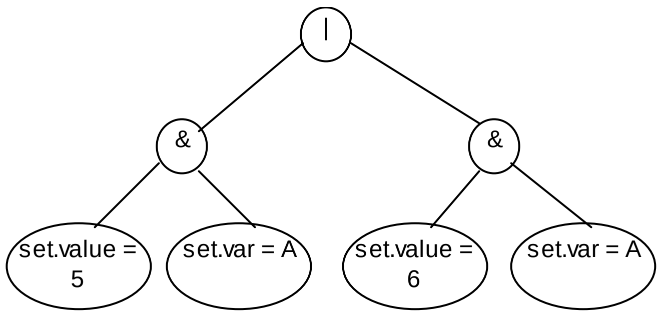


Figure 6-11 another tree view representing an expression in DNF

Step 4: we get back the expression from the tree which is constructed in step 3. In our example, the expression we get in this step is ‘set.value = “5” & set.var = “A” | set.value = “6” & set.var = “A”’, which is in DNF.

Following the above four steps, we obtain an expression in DNF.

- Construction

We design a data structure, called DP (Debugging Point) to hold specification of a debugging point. The attributes and constructors as well as their descriptions of DP class are shown as follows:

```
Class DP {
    List target;
    List modifier;
    int type;
    int line;
    //this constructor is invoked if the debugging point is with XVCL command type
    public DP (List target, List modifier, int type);
        // this constructor is invoked if the debugging point is with line type
        public DP (List target, int line, int type);
  }
```

Field description:

target – specifies a list of x-frames in which the debugging point is set

modifier – specifies the description of a debugging point. This attribute is only for debugging points with XVCL command type.

type – indicates the type of a debugging point. 0 is for line type and 1 is for XVCL command type.

line – indicates the line at which the debugging point is set. This attribute is only for debugging points with line type.

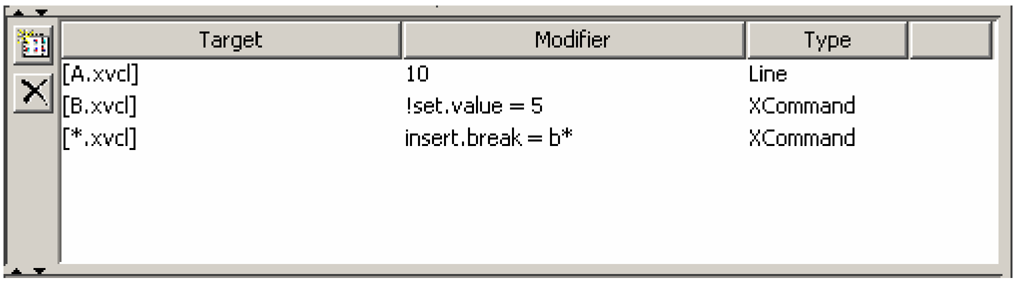
Once a DP object for a debugging point is constructed, IXP can interpret this debugging point.

6.3.3 Debugging point table

When a debugging point is set, IXP need keep it till the end of processing. Therefore, we need a data structure to contain all debugging points that are currently set. DPT (Debugging Point Table) is designed for this purpose.

DPT is more than a container of DP (Debugging Points) objects. It provides operations for the user to manipulate debugging points currently kept in the table. The user can insert a debugging point into DPT. If the user finds a debugging point in the table is not suitable or it is not useful any more, he can remove it from table. In addition, if the user finds some errors in a debugging point, he may choose to modify it.

IXP provides Debugging Point Window which shows all debugging points that are currently contained in DPT.



Target	Modifier	Type	
[A.xvcl]	10	Line	
[B.xvcl]	!set.value = 5	XCommand	
[*.xvcl]	insert.break = b*	XCommand	

Figure 6-12 Debugging Point Window

The user can find all DPT operations in Appendix A.

6.3.4 Stepping functions

As we described in Chapter5, IXP provides four different modes of stepping functions: step through, step out, step over and skip over. The user can switch between stepping mode at any time during the processing, which would affect subsequent stepping. In our implementation, we use four variables to control the behavior of these stepping functions.

- Step Through

The Boolean variable "step_thr" is used to control the behavior of Step Through function. By default, "step_thr" is false. Once the user clicks the Step Through button and chooses to step through the source codes, "step_thr" is set to true. In the subsequent processing, IXP will suspend at each XVCL command since the variable "step_thr" is true. If the user clicks some other buttons (e.g. Step Out, Step Over, Skip Over and Run) later, “step_thr” will be set back to false. In the subsequent processing, IXP will switch to another mode of stepping depending upon which button the user has clicked.

- Step Out

The Boolean variable “step_out” is used to control the behavior of Step Out function. By default, “step_out” is false. Once the user clicks the Step Out button, “step_out” is set to true. And the parent structure of the current structure in which the user clicks the Step Out button is kept by IXP as well. To describe how IXP works in Step Out mode, we present the following algorithm.

```
stepouts:= parent structure of the current one in which the user clicks Step Out button;
currents:= XVCL structure in which the current processing x-command is;
if (a debugging point is set or an error occurs)
    IXP suspends at the current processing XVCL command
else if (currents equals stepouts)
    IXP suspends at the current processing XVCL command
else
    IXP process the XVCL command but does not suspend at it
```

If the user clicks some other buttons later, the variable “step_out” will be set back to false and “stepouts” will be set to null. In the subsequent processing, IXP will switch to another mode of stepping depending upon which button the user has clicked.

- Step Over

The Boolean variable “step_over” is used to control the behavior of Step Over function. By default, “step_over” is false. Once the user clicks the Step Over button, “step_over” is set to true. IXP uses a variable “stepoverf” to hold the current x-frame in which users clicks the button. We use the following algorithm to describe how IXP works in Step Over mode.

```
stepoverf: = x-frame in which the user clicks Step Over button;
currentf:= x-frame in which the current processing x-command is;
if (a debugging point is set or an error occurs)
        IXP suspends at the current processing XVCL command
else if (currentf equals stepoverf)
        IXP suspends at the current processing XVCL command
else if (currentf is an ancestor frame of stepoverf)
        IXP suspends at the current processing XVCL command
else if (currentf is a descendant frame of stepoverf)
        IXP process the XVCL command but does not suspend at it
```

If the user clicks some other buttons later, the variable “step_over” will be set back to false and “stepoverf” will be set to null. In the subsequent processing, IXP will switch to another mode of stepping depending upon which button the user has clicked.

- Skip Over

The Boolean variable “skip_over” is used to control the behavior of Skip Over function. By default, “skip_over” is false. Once the user clicks the Skip Over button, “skip_over” is set to true. In the subsequent processing, IXP will suspend at each XVCL command. But, if IXP encounters a <adapt> command, it will suspend but won’t process this command. That means IXP will not process the adapted x-frame. If the user clicks some other buttons later, the variable “skip_over” will be set back to false. In the subsequent processing, IXP will switch to another mode of stepping depending upon which button the user has clicked.

}

6.3.5 Outputs formatting

In IXP, partially emitted outputs are shown in the Output Window. As described in the section 5.4.9, the user can set the formatting of the outputs in the following ways:

- Show only recent N output lines in the Output Window when the number of the output lines reaches M.

  Here, M is defined by “maximum number of output lines” and N is defined by “minimum number of output lines”. See section 5.4.9 for details.

- Show from which x-frame the output comes
- Show to which output file the output goes

In our implementation, an output line is defined as an OutputLine object which keeps necessary information. The attributes and constructors as well as their descriptions of OutputLine class are shown as follows:

```
class OutputLine {
        String from; // from which x-frame this line of outputs comes
        String to; //to which output file this line of outputs goes
        String output; // content of this line of outputs
        OutputLine next; //next node in the circular linked list

        public OutputLine (String f, String t, String o) {
            from = f;
            to = t;
            output = o;
        }
}
```

We use an N-length circular linked list to hold the output objects. When IXP processes a line of outputs, it will insert this OutputLine object to the list. This circular structure ensures that only recently produced N output lines are kept.

IXP uses a variable, named “maxLineNum” to hold the information of “maximum number of output lines” and a static variable “chainCount” to count the number of lines of outputs currently shown in the Output Window. If “chainCount” is large than or equal to “maxLineNum”, the Output Window will refresh and “chainCount” will reset to be 0.

IXP uses two Boolean variables “showfrom” and “showto” to control the format of the outputs. Both are false by default. If “showfrom” is set to be true, the Output Window will show the Output Window will show which parts of the output comes from which x-frame. If “showto” is set to be true, the Output Window will show which parts of the output goes to which output file. The user is free to alter the value of “showfrom” and “showto” during the processing in order to get the different view of the outputs.

6.3.6 Other modifications made to the batch XVCL processor

To meet the specifications of IXP, we should make some other modifications to the batch XVCL processor.

6.3.6.1 Implement XVCL main process as an interruptable thread

IXP can suspend processing of x-framework whenever an error or a debugging point is encountered. In our implementation, we extend XVCL class (main class of the program) from *java.lang.thread* Class.

public class XVCL extends Thread {……}

To suspend x-framework processing, we use the following codes:

```
try {
            System.out.println("suspend");
             XVCLProcess.xvcl.suspend();
  }catch (Exception e){}
```

XVCLProcess.xvcl is the object that holds the reference to XVCL main process (e.g. An XVCL class object).

To resume the main process, we use the following code:

```
XVCLProcess.xvcl.resume();
```

In this way, IXP may suspend and resume the processing of x-framework anywhere as the user expects.

6.3.6.2 Check whether an XVCL command is a debugging point

When IXP encounters an XVCL command, after parsing it, IXP need check whether this command is a user designated debugging point.

IXP checks through DPT (Debugging Point Table) till a matching debugging point is found. The algorithm to determine whether a debugging point is set at an XVCL command is described as follows:

Step 1: Get a debugging point from DPT in a sequential manner. Terminate the procedure if there is no more debugging point in DPT.

Step 2: Check whether the x-frame, in which the current processing x-command is, is contained in the list of x-frames in which this debugging point is set. If not, this debugging point is not set at the current processing XVCL command. We go back to step 1. Otherwise, we proceed to step 3.

Step 3: Retrieve the type of the debugging point (line type or XVCL command type).

If it is with line type, we compare the line number the debugging point specifies with the line number of the current XVCL command. If two numbers are the same, we can say a debugging point is set on the current XVCL command and we terminate checking procedure. Otherwise, we go back to step 1.

If it is with XVCL command type, we check whether the specification of the debugging point matches the definition of current XVCL command. If yes, we can say a debugging point is set on the current XVCL command and we terminates checking procedure. Otherwise, we go back to step 1.

We use an example to illustrate how this algorithm works.

Suppose the current processing XVCL command is <set var = "X" value = "5"/> with line number "18" in x-frame A. The current Debugging Point Table is shown as below:

	Target	**Modifier**	**Type**
	B.XVCL	**18**	**line**
	A.XVCL	**set.var = Y**	**XVCL command**
→	**A.XVCL**	**set.var = X & set.value = 5**	**XVCL command**

Figure 6-13 an example of Debugging Point Table

Following the algorithm, we can draw the following conclusions:

The first debugging point is not set at the current x-command, since it is set in x-frame B.

The second debugging point is also not set at the current x-command, since its specification doesn't match the definition of the current x-command.

The last debugging point in DPT is set at the current x-command. It is set in x-frame A, and its specification matches the definition of current x-command. Therefore, we can say a user designated debugging point is set at the current processing XVCL command.

If a debugging point is set at the current XVCL command, IXP suspend the processing of x-frameworks at this command. Otherwise, IXP processes this XVCL command in the same way as the batch XVCL processor does.

6.3.6.3 The user interface

IXP is a graphical system. A friendly user interface is developed to ease users' jobs. In our implementation, we add one more subpackage, named "frames", to the batch XVCL processor. This subpackage contains UI classes, such as "Source Window", "Variable Window", "Edit Variable Dialog" and etc. Now, the package "xvcl" contains three subpackages in total: "commands", "util" and "frames". Figure 6-14 shows the "client" relationship among the three subpackages.

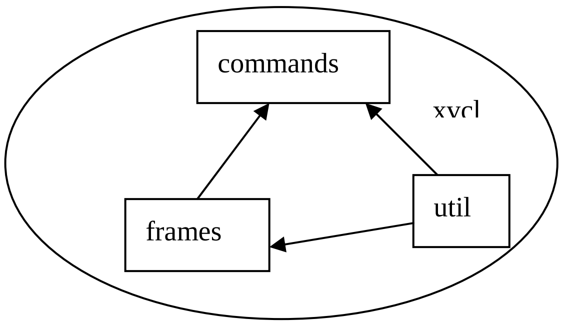


Figure 6-14 the "client" relationship among three subpackages

Subpackage "commands" is a client of "util", since classes in the former may import some classes in the latter. Following this way, "command" is also a client of "frames" and "frames" is a client of "util".

Chapter 7

Conclusions and future work

As a language for configuring variants, XVCL provides an effective way to create an implementation of generic product line architectures. XVCL is based on the frame technology, expressed with XML conventions. XVCL is supported by a tool – the XVCL processor. XVCL processor interprets XVCL commands and automates the customization process.

In this thesis, we presented an interactive, debugging environment for XVCL – Interactive XVCL Processor, or IXP for short. IXP extends the batch XVCL processor with dynamic analysis and debugging features, allowing users analyze x-frameworks at runtime. We built IXP on top of the batch XVCL processor.

XVCL is a static meta-programming system. By static meta-programming we mean systems that work at the program construction-time, before compilation (as opposed to dynamic meta-programming systems in which programs are modified at runtime). Analysis and debugging of meta-programs is a difficult and mostly unexplored problem. While there are similarities between analysis/debugging of conventional programs and meta-programs, the analogy very soon breaks. A meta-program is a generic program (or a generic program architecture as in case of XVCL) from which many concrete programs can be generated. Therefore, analysis/debugging of a meta-program must primarily concentrate on the generic aspects and the generation process rather than on traditional analysis/debugging of program behavior.

The main contribution of this Thesis is in setting up a stage for automated analysis/debugging of static meta-programming systems. Based on general requirements for analysis/debugging of meta-programs, we developed an analysis/debugging method and tool, called IXP, for meta-programming in XVCL. While the details of analysis/debugging methods depend on a specific meta-programming technique, we believe our results are interesting and useful in general.

Unique features of IXP include:

- the ability to trace the program generation process in a flexible way. We developed a specification language allowing a user to specify debugging points (a debugging point is a point at which the processing suspends allowing a user to inspect the state of processing)
- the ability to inspect and modify values of XVCL variables at the debugging points
- the ability to define undefined meta-components which allows the user to process incomplete meta-programs
- the ability to step through a meta-program in multiple ways, as required to understand the generic nature of a meta-program
- the ability to store useful trace information resulting from processing for further analysis

In the thesis, we described functionalities of IXP as well as its design and implementation. IXP extends the batch XCVCL processor in the following ways:

- includes a modified symbol table that can be viewed and changed during processing
- includes the debugging points specification via the DPSL (Debugging Points Specification Language)
- includes a debugging point table that holds all debugging points that are currently set
- includes four stepping operations
- includes output formatting functions
- includes some other modifications made to the implementation of the batch XVCL processor

The following extensions of this project should be considered in the future:

- Integrate an X-Frame Query Language (so-called XFQL) into IXP to further enhance analysis features of IXP. Prototypes of such a language have been developed in other projects, however, their usefulness was limited by lack of the environment provided by IXP

  XFQL is a static analysis tool for x-frameworks. With XFQL, the user can query x-frameworks. The example of such a query is "how many variables are defined in x-frame A". Currently, XFQL assumes the x-framework that it queries is complete. That is to say, there is no undefined variable or undefined x-frame in the x-framework. But in real life, that may be some incomplete x-frames under development. XFQL does not work well with incomplete x-frames. With IXP, we can improve XFQL. When a user's query involves a variable (or an x-frame) which is undefined, IXP will suspend the processing of x-frameworks and provide the user an interface to feed a value (or designate a path) for this variable (or x-frame). Once the variable (or x-frame) is defined successfully, IXP resumes the processing and XFQL processes the query normally.

- Integrate IXP with SGE (Smart Graphical Editor)

  SGE is the editor for XVCL. SGE shows x-frames in an easy to grasp, XML-free format and provides browsing features. In IXP, the Source Window is not editable. To enhance the IXP's functions, in the future, we may replace the Source Window with SGE. In that way, when the processing of x-frameworks suspends, the user may even edit the x-frames (e.g. modify the Texture Contents). And those changes will take effects when the processing resumes.

- Currently, IXP suspends processing of x-framework at an XVCL command only. We may remove this restriction in future. We allow x-framework processing to suspend at every line of an x-frame, no matter it is an XVCL command or Texture Content.

- In the current implementation, with DPSL, a debugging point with XVCL command type is defined as "[frame-list].specification of an XVCL command". "frame-list" defines a list of frames in which this debugging point is set. A "frame-list" is specified by giving a list of x-frame names, such as "[A, B, C]". Such a specification is static. In future, we want to extend the specification to be dynamic. The dynamic specification of "frame-list" doesn't provide names of x-frames directly. Instead, it presents the "frame-list" in a dynamic way. ["All x-frames that are adapted by x-frame A"] is an example of such a dynamic specification of "frame-list", which means the x-framework processing only suspends in x-frames which are adapted by x-frame A.

- Currently, IXP is a separate system from the batch XVCL processor although it is extend from the latter. To invoke IXP, we type the command "java ixp.XVCL 'SPC name'" in the command line. To invoke the batch XVCL processor, instead, we type the command "java xvcl.XVCL 'SPC name'". This situation is messy. To improve it, we will try to embed IXP as one of XVCL processor's options in the future. In that way, to invoke IXP (in fact, the debug mode of the XVCL processor), we type "java xvcl.XVCL **–D** 'SPC name'" in the command line. Without "-D" option, the normal processor is invoked instead.

References:

[1] Adrian, A. and Charles, G. (2003). Part I: Introduction to software life-cycle. Lecture Notes of CS22L Software Engineering I, University of the West Indies, Cave Hill Campus, 2003.

[2] Bassett, P. (1997). Framing Software Reuse – Lessons from Real World, Yourdon Press, Prentice Hall.

[3] Clements, P. and Northrop, L. Software Product Lines: Practices and Patterns Addison-Wesley 2002.

[4] Cxref Homepage. C Program Cross-Reference, Retrieved from the World Wide Web: http://www.gedanken.demon.co.uk/cxref/

[5] Jarzabek, S. and Zhang, H. (2001). XML-based method and tool for handling variant requirements in domain models. In Proceedings of Fifth IEEE International Symposium on Requirement Engineering (RE'01), Toronto, Canada, August 2001.

[6] Jarzabek, S., Shen, R. and Sun, Z. (2002). Understanding meta-programs and product line architectures. School of Computing, National University of Singapore, Singapore. January, 2002

[7] JSwat. A graphical Java debugger, Retrieved from the World Wide Web: http://www.bluemarsh.com/java/jswat/index.html

[8] Li, S. (2002). JDOM based XVCL processor. Research Presentation, School of Computing, National University of Singapore, Singapore. Jun, 2002

[9] Netron Fusion^{TM} (2004). Netron Fusion^{TM} version 3.4 Technical Overview, 2004.

[10] Parnas, D. On the Design and Development of Program Families, IEEE Trans. On Software Eng., March 1976.

[11] Soe, M.S. (2002). XML-based language and tool for handling variants in software product lines. M.Sc. Thesis, School of Computing, National University of Singapore, Singapore. 2002

[12] Soe, M.S., Jarzabek, S. and Zhang. H. (2002). XVCL: a tutorial. Research Report, School of Computing, National University of Singapore, Singapore. Feb, 2002

[13] Stylus Studio (2001). Stylus Studio XSLT debugger, Retrieved from the World Wide Web: http://www.stylusstudio.com/xslt_debugger.html

[14] Wong, T.W., Jarzabek, S., Soe, M.S., Shen, R. and Zhang, H. (2001). XVCL implementation of frame processor. In Proceedings of Symposium on Software Reusability (SSR' 01), Toronto, Canada, May 2001.

[15] XSLT (1999). W3C XSL Transformations (XSLT) version 1.0, Retrieved from the World Wide Web: http://www.w3.org/TR/xslt

[16] Zhang, H. (2001). XML-based Variant Configuration Language (XVCL): Concepts and Examples. Research Report, School of Computing, National University of Singapore, Singapore. 2001

Appendix A: Documentation of important classes

ADT of VariableInfo class

class VariableInfo extends Variable {

Overview: VariableInfo is a data structure holds variable information. VariableInfo extends from Variable class.

Attributes area:

private History valuehistory; // a list
private boolean modified;
private boolean inserted;
private String path;

Public interface:

Operation header: public VariableInfo(History v, boolean i)
Description: Constructor for VariableInfo

Operation header: public void setPath (String s)
Description: Set ‘path’ to s

Operation header: public void setModified (boolean m)
Description: Set ‘modified’ to m

Operation header: public void setInserted (boolean i)
Description: Set ‘inserted’ to i

Operation header: public void setHistory (History h)
Description: Set ‘variablehistory’ to h

Operation header: public String getPath()
Description: Retrieve ‘path’ of this variable where the variable is defined.

Operation header: public boolean getModified()
Description: Retrieve ‘modified’ flag of this variable.

Operation header: public boolean getInserted()
Description: Retrieve ‘inserted’ flag of this variable.

Operation header: public History getHistory ()
Description: Retrieve ‘valuehistory’ of this variable.

Operation header: public void insertAtHead ()

Description: Insert the current variable value at the head of 'valuehistory'

Operation header: public void removeHead ()
Description: Remove the head element of list 'valuehistory'

Operation header: public String getNewestValue ()
Description: Return the newest value of this variable (current value of the variable).

Operation header: public void restoreToPrevious ()
Description: Restore the value of a variable to its previous value.

Operation header: public int varListSize ()
Description: Return the size of 'valuehistory'.

Operation header: public String toString ()
Description: Returns a string representation of the object.

}

ADT of DPT class

class DPT {
Overview: DPT is data structure used to hold a set of debugging points. It provides some functions for IXP to manipulate the set of debugging points.
Attribute area: static List DPT;

Public interface:

Operation header: public static void insert (DP dp)
Parameters: 'dp' is a user specified debugging point
Description: This operation inserts a debugging point into the table

Operation header: public static void delete (int index)
Parameters: 'index' is index of a debugging point in the table, which user wants to delete. If index is out of range, just ignore.
Description: This operation deletes a debugging point from the table

Operation header: public static void modify (int index, List newmodifier);
Parameter: 'index' is index of a debugging point in the table, which user wants to modify.
Description: This operation modifies a debugging point in the table
Notice: This method is invoked only if the type of the debugging point is XVCL command

Operation header: public static void modify (int index, int newline);
Parameter: 'index' is index of a debugging point in the table, which user wants to modify.

Description: This operation modifies a debugging point in table
Notice: This method is invoked only if the type of the debugging point is line

Operation header: public static int size ();
Description: Return size of current debugging point table

Operation header: public static DP get (int index);
Parameter: 'index' is index of the debugging point in the table, which user wants to retrieve
Description: Retrieve a debugging point from debugging point table

Operation header: public static void setDPT (List l);
Description: Set debugging point table to specified one

Operation header: public static List getDPT ();
Description: Retrieve the current debugging point table

Appendix B: Summary of XVCL commands

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 x-frame texts, • 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 breakpoint in an x-frame.	The <break> command marks a breakpoint (slot) 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>**	**break**: defines the name of the breakpoint.	The <insert> command replaces the breakpoint “*break-name*” in the adapted x-subframework with the *insert-body*.

<insert-before break = **”***break-name*”**>** *insert-body* **</insert-before >** **<insert-after break=”***break-name***”>** *insert-body* **</insert-after >**		The <insert-before> command inserts the *insert-body* before the breakpoint “*break-name*” in the adapted x-subframework. The <insert-after> command inserts the *insert-body* after the breakpoint “*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 single-value variable. **value**: defines the value to be assigned.	The <set> command assigns a “*value*” defined in the “value” attribute to single-value variable “*var-name*” defined in the “var” attribute.
<set-multi var=”*var-name***”** **value=”***value1, value2, …*” />	**var**: defines the name of 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 the “value” attribute to a multi-value variable “*var-name”* defined in the “var” attribute.
<value-of expr = ”*expression***”** />	**expr**: defines an expression to be evaluated.	The value of the “*expression*” is evaluated and the result replaces the <value-of> command.
<select option = ”*var-name*”**>** *select-body*: may contain options listed below **</select>**	**option**: The “option” attribute in <select> command defines the variable whose value will be matched in <option> commands.	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 the value of “*var-name*” matches

select-body: **<option-undefined>** (optional) *option-body* **</option-undefined>** **<option value = ”***value***”>** (0 or more) *option-body* **</option>** **<otherwise>** (optional) *option-body* **</otherwise>**	**value**: The “value” attribute in <option> command defines the value to be matched.	<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 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 single-value variable. The *while-body* may contain a mixture of textual content and XVCL commands.
<ifdef var=”*name***”>** *if –body* **</ifdef>**	**var:** name of a variable that may or may not in Symbol Table	If variable “*name*” is defined (that is, it exists in the Symbol Table), the XVCL processor processes the if-body. Otherwise, the if-body is not processed.
<ifndef var=”*name***”>** *if –body* **</ifndef>**	**var:** name of a variable that may or may not in Symbol Table	If variable “*name*” is undefined, the XVCL processor processes the if-body. Otherwise, the if-body is not processed.

<remove var="*name***">**	**var:** a variable name to be removed from the Symbol Table.	This command un-defines a variable "name" described in *"var"* attribute by removing it from the Symbol Table.
<message text= "*message*" **/>**	**text:** specifies an error message.	When the XVCL processor encounters the <*message*> command, it will display an error message on the screen.
<!-- comment -->		Text enclosed between <!-- --> is considered a comment. Comments may spread over multiple lines.