XML Implementation of Frame Processor

Tak Wai Wong, Stan Jarzabek, Soe Myat Swe, Ru Shen, Hongyu Zhang

Department of Computer Science

School of Computing

National University of Singapore

Singapore 117543

+65 874 2863 (phone), +65 779 4580 (fax)

{wongtakw, stan, soemyats, shenru, zhanghy}@comp.nus.edu.sg

ABSTRACT

A quantitative study has shown that frame technology [1] supported by Fusion™ toolset can lead to reduction in time-to-market (70%) and project costs (84%). Frame technology has been developed to handle large COBOL-based business software product families. We wished to investigate how the principle of frame approach can be applied to support product families in other application domains, in particular to build distributed component-based systems written in Object-Oriented languages. As Fusion™ is tightly coupled with COBOL, we implemented our own tools based on frame concepts using the XML technology. In our solution, a generic architecture for a product family is a hierarchy of XML documents. Each such document contains a reusable program fragment instrumented for change with XML tags. We use a tool built on top of XML parsing framework JAXP to process documents in order to produce a custom member of a product family. Our solution is cost-effective and extensible. In the paper, we describe our solution, illustrating its use with examples. We intend to make our solution available to public in order to encourage investigation of frame concepts in other application domains, implementation languages and platforms.

Categories and Subject Descriptors

D.2.13 [**Reusable Software**]: Domain Engineering

General Terms

Design, Experimentation, Languages

Keywords

Frame technology, XML, product families, variant requirements

1. INTRODUCTION

The key issue in domain engineering approach is to identify and represent commonality and variability among the members of a product family. This must be done at the product family requirements, generic architecture and architecture implementation levels. A *generic product family architecture* defines an architecture shared by product family members and provides components to be reused across the members. Both architecture and its components must be flexible, to facilitate accommodation of variants into a specific product.

Software engineering techniques to handle variants in product families can be broadly classified into two categories, namely program construction time techniques (such as pre-processing, configuration management, parameterization, OO inheritance, templates, meta-programming, generation techniques and frame technology) and runtime techniques (such as dynamic binding, late binding and design patterns). The decision whether a given variant should be handled during program construction or during runtime is an important one as it affects product attributes such as usability, performance, complexity of code (therefore maintainability) and many other product attributes. We refer the reader to [6] for detailed discussion and comparative study of both categories of techniques for handling variants. In this paper, we are interested in handling variants during program construction time.

Among construction-time techniques, frame technology [1] has particularly good record in industrial applications. A quantitative study has shown that frame technology can lead to reduction in time-to-market (70%) and project costs (84%). It can also dramatically improve the effectiveness of project teams. Percentage reuse can vary from 50% to about 95%. This data results from independent assessment by consulting firm QSM, based on project benchmark database including productivity data from thousands of software projects using a wide range of techniques [1]. These productivity gains are due to flexibility of frame-based architectures and their ability to evolve over years.

Excellent record of frame technology in large scale industrial applications encouraged us to explore frame approach in our domain engineering projects. In early projects, we applied frames to component-based product families in facility reservation system domain [4], [10]. In the current project, we are developing methods to support a family of distributed Computer Aided Dispatch systems (CAD for short). CAD systems are used by police, fire & rescue, health and port. Basically, CAD systems support dispatch of units to incidents (e.g., a police unit to the place of car accident). However, the specific context of the operation (such as police or fire & rescue) results in many variations on the basic operational theme that yield a product family.

We wished to investigate if frame principles could be applied to distributed component-based product families and mission-critical systems such as CAD, bringing similar productivity gains as in COBOL business systems. We were also interested in how frame approach blended with component-based systems, component platforms, Object-Oriented analysis/design methods for distributed applications and programming languages such as Java.

However, Netron's Fusion™ toolset is tightly coupled with COBOL, in terms of both tool environment and frame command language. Most of the framing tools in Fusion™ work only with COBOL and run on top of Micro Focus™ environment. Frame language commands are influenced by COBOL and do not address contemporary design methods, language features and platforms (such as Object-Oriented methods, component-based design and distributed component platforms). CAD systems are distributed applications, running on top of the platforms supporting a subset of CORBA services for component communication. It had soon become clear that in order to conveniently apply frame concepts to CAD product family, we need to decouple frame processor from COBOL and enhance frame concepts to address specific needs of our project.

Therefore, we decided to implement our own language and tool, based on essential frame concepts, but free of COBOL inheritance and including extensions that we required. We chose the XML technology for the implementation. In our solution, a generic architecture for a product family is a hierarchy of XML documents. Each such document, called an x-frame, contains a program fragment instrumented for change with XML tags – XML counterparts of frame commands. XML tags mark variation points at which x-frame can be customized to fit specific reuse context. Customization is done by a tool that we built on top of XML parsing framework JAXP.

Our solution is cost-effective and extensible. XML standards and widely available XML parsers (such as JAXP) offer ample room to implement many other tools (such as documentation tools, browsers, smart editors, program transformation and reverse engineering tools) that can further support the product family. With permission of Netron™, we intend to make our solution available to public and encourage wide investigation of frame concepts in order to accumulate experiences from different types of software projects.

In the paper, we describe the principles of the frame approach and our XML solution, illustrating with examples from our project on engineering a CAD product family.

2. RELATED WORK

Approaches to supporting product families differ in types of software assets that are reused, in the process that leads to discovery and design of reusable assets and in the role reusable assets play in building individual software systems, members of a family. As early as in 1970's, Parnas [14] proposed modularization criteria and information hiding for handling variants in product families. Object-Oriented frameworks [11] use information hiding along with inheritance, dynamic binding and design patterns [8] to handle variants in a product family. The vision of component-based software engineering and distributed component platforms [3] is also to support product families.

An application generator approach is a powerful and cost-effective solution to the product family problem in well understood and stable domains [2]. Program generation may be accomplished by transformation or assembly of generic program parts (a combination of transformation and assembly process is also feasible). Meta-programming [6] and program transformation systems customize a program to meet specific variant requirements at program construction time before compilation.

Macros are probably the oldest example of meta-programming. Macro processors have been applied to both procedural and Object-Oriented languages [5], however they have never been accepted as an effective technique to handle variants in product lines [12]. Frame technology is a form of advanced macro system [1]. It is therefore both intriguing and worth effort to investigate the principles that have made frame technology so successful. This was the main motivation for undertaking the work described in this paper.

Many computational aspects of a program spread throughout the whole program and cannot be nicely confined to a small number of program components. Examples include business logic, platform dependencies or codes that have to do with system-wide qualities such as performance and fault tolerance. In aspect-oriented programming [13], each computational aspect is programmed separately and a mechanism is provided to compose aspects into an executable program. In multi dimensional hyperspace approach [17] the concept of separation of concerns is extended to levels of design and analysis. The essence of both aspects and hyperspaces is the ability to understand and manage variants within certain aspects/concerns in separation from others, as much as it is possible. Both approaches give raise to a product family.

Other approaches to supporting product families include table-driven techniques, parameterized programming [9], configuration management and PCL [15]. It is beyond the scope of this paper to discuss these approaches in detail. We refer the reader to [6] for an excellent comparative study of various approaches to handling product families, both at construction-time and runtime.

3. DESCRIPTION OF COMPUTER AIDED DISPATCH (CAD) DOMAIN

A Computer Aided Dispatch (CAD for short) product family will serve as illustration of our XML-based language and tool for framing. Figure 1 depicts a basic operational scenario, roles and elements of a CAD system for Police.

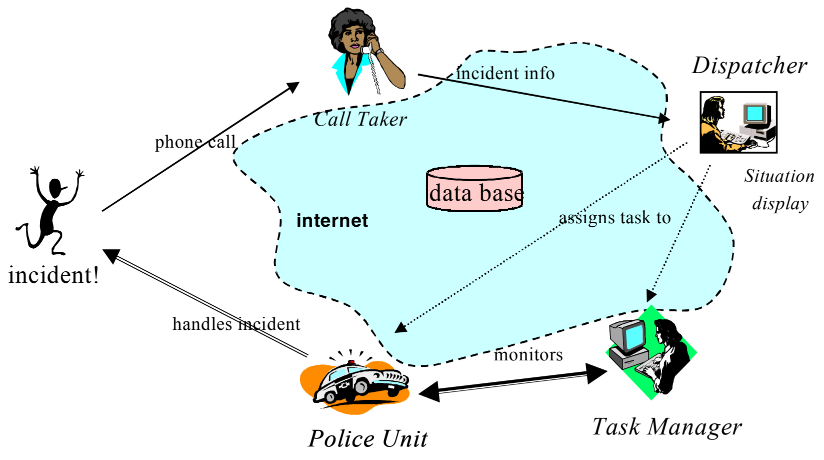

Figure 1. An operational scenario in CAD system for Police

A Call Taker receives information about an incident and informs a Dispatcher about the incident. The Dispatcher examines the "Situation Display" that shows a map of the area where the incident happened. Then, the Dispatcher assigns a task of handling the incident to the Police Unit, for example, this might be a police car that is closest to the place of incident. The Police Unit approaches the place of incident and handles the problem. The Police Unit informs the Task Manager about the progress of action. The Task Manager monitors the situation and at the end - closes the case. The information about current and past incidents is stored in the database.

The specific context of the operation (such as police or fire & rescue) results in many variations on the basic operational scheme. For example, CAD systems differ in rules of how resources are assigned to tasks, monitoring, reporting and timing requirements, specific information to be stored in a database, system component deployment strategies, reliability and availability requirements, and so on. These commonalties and variants give raise to a product family. In a product family approach, we exploit commonalties among CAD systems and engineer CAD systems from a common base of reusable software assets that include domain models (i.e., requirements for CAD product family), generic architecture and its implementation, test cases, etc.

In practice, companies never deal with arbitrary variants across product family members. During product family scoping [7], a company identifies variants that have high business value and are worth domain engineering effort. In this paper, we do not elaborate further on the product family scoping, but we should mention that variants used in examples in the following sections are considered important by the industrial partner involved in our project.

4. AN OVERVIEW OF THE FRAME APPROACH

In the frame approach [1], reusable program parts are called frames. A frame is instrumented with *frame commands* that show how the frame can customize its own contents as well as contents of other frames. The customizable contents of a COBOL frame typically contains a program, section, paragraph or declarations. Frame commands make frames customizable by allowing one to select pre-defined options (i.e., anticipated variants) based on certain conditions, marking breakpoints where unexpected variants can be inserted and providing variables as a parameterization mechanism.

Frames are building blocks of a product family generic architecture. Frames are organized into a hierarchy. The frame hierarchy results from the process of turning a set of separate programs that display much commonality into a product family. Frames related to different computational aspects are grouped together. For example, frames related to the user interface, business logic, database or platform-dependent code form sub-hierarchies (or layers) in a frame-based generic architecture. A frame hierarchy is designed to explicate and localize (whenever possible) the impact of change in order to simplify addressing variants. Frames contain knowledge of how to address anticipated variants. There is also a provision for addressing unexpected new requirements, which is essential in poorly understood and evolving domains.

Each member of a product family is characterized by certain anticipated variants and possibly some unexpected new requirements. Construction of a product family member is done by frame customization and assembling process. The root of a frame hierarchy, called the *specification frame* (SPC), describes all customization required to obtain a specific member of a product family. The root is where the most context sensitive options are selected. Defaults are used automatically if no option is selected explicitly. This reduces the amount of details a frame higher in the hierarchy has to handle. In general, the lowest frames in the hierarchy are the most context-free, and thus, most reusable. Context-sensitivity increases (while reusability tends to decrease) up the hierarchy towards the root.

The construction process, executed by frame processor, uses depth-first traversal of a frame hierarchy (Figure 2). The process starts with the root specification frame SPC (frame A in Figure 2). During this traversal, frame processor emits the contents of each visited frame. At the same time, it carries out customization instructions from the frame commands. Frames are assembled into a single output file. This construction process incorporates variants to produce a specific member of a product family that meets those variants. The output file contains customized source code for that member.

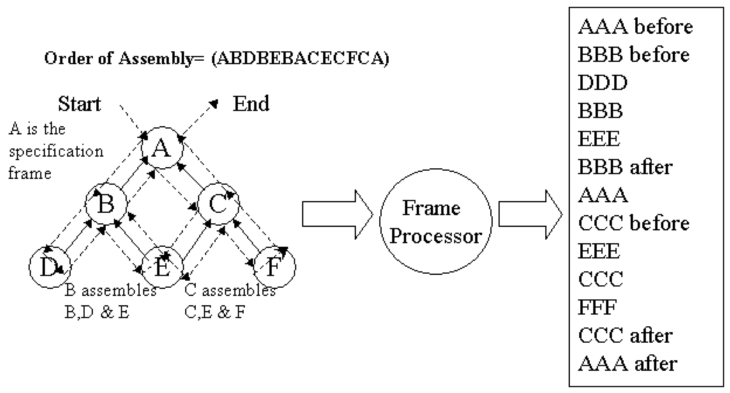

Figure 2. Frame processing

We shall now illustrate the construction process. Figure 3 describes the contents of SPC A and frames B and C. The contents of bottom level frames D, E and F is simply DDD, EEE and FFF, respectively.

```
.%SPC A includes the following text and COPY commands;
.%COPY X includes the contents of frame X
AAA before
.COPY B .
AAA
.COPY C .
AAA after
-------------------------------------------------------
.%frame B includes the following text and COPY commands:
BBB before
.COPY D .
BBB
.COPY E .
BBB after
-------------------------------------------------------
.%frame C includes the following text and COPY commands:
CCC before
.COPY E .
CCC
.COPY F .
CCC after
```

Figure 3. The contents of SPC A (at the top of frame hierarchy in Figure 2), frame B and frame C.

The construction process produces the result indicated on the right hand side of Figure 2. So far, we explained only the assembly process. The output from each visited frame can be customized during the frame assembly process. This customization is specified by frame commands. In the following section, we describe frame commands using XML notation. We also show examples of extensions that we found useful in our project.

Frames are spawned from macros, but frame mechanism provides a better control over changes and a more flexible parameter scoping and parameter passing mechanism than most of other macro systems. "Conventional macros use "flat" scoping… all parameters are global… they can be set and reset by all others… Frames parameters, on the other hand, are local to the frame that first sets their values… This means they can be set and reset only within the defining frame; descendent frames can only read them… when a defining frame exist, all its parameters become undefined… Conventional macros have a "daisy-chaining" parameter passing convention. That is, if macro X invokes Y and Y invokes Z, parameters passed from X to Z, must either pass through Y or must be global… forces macro Y to be more context sensitive... if frames X, Y and Z invoke each other as above, X parameterizes Z without involving frame Y… macros find it hard to modify the bodies of other macros... frames can, if necessary, directly rework – remove or replace – any parts of a frame that don't fit the local context." [1]

5. FRAME CONCEPTS IN XML

5.1 Overview of XML and JAXP

Extensible Markup Language (XML) separates the description of data from its interpretation [18]. XML uses *tags* to mark the beginning and end of the data. The tags can also contain *attributes* that provide additional information for tagged data.

The *Java API for XML Parsing* (JAXP) is a framework that supports parsing of XML documents [16]. As the parser scans through an XML document, it generates an event for each occurrence of the beginning of a tag, end of tag, data, processing instructions, etc. For each of those events, we can write a custom *event handler method*. Whenever the parser generates an event, a corresponding event handler method is invoked to interpret it.

```
<message to="you@your.com" from="me@my.com"
subject="XML Demo">
… text of the message
</message>
```

In the above example, the parser first invokes an event for the tag message with its attributes to, from and subjects. Next, the parser invokes an event for text of the message and, finally, for end of message. It is up to the event handler methods to interpret XML documents.

XML document may include *processing instructions* that cater for any additional processing requirements. Processing instructions have the following format: <?target instructions?> where the *target* is the name of a processor and *instructions* contain commands for the processor. This feature opens endless possibilities for using XML in many application contexts, in particular to implement programming tools.

5.2 Frames as XML Documents

In our solution, a generic architecture for a product family becomes a hierarchy of inter-related XML documents, called x-frames. An x-frame is an XML counterpart of a frame in Fusion™. The customizable contents of an x-frame is typically a function, class, component, interface or a code fragment such as a block of declarations. This content conforms to XML standards, with XML tags marking variation points. Like a frame, an x-frame is an adaptable reusable asset, a building block of a generic architecture for a product family.

Figure 4 depicts an excerpt from an x-frame architecture for the CAD product family. When the processor encounters a COPY command, it creates a copy of a specified x-frame and applies customization commands that may be (optionally) specified after the COPY command. The reader will find further details of the customization commands in the next section.

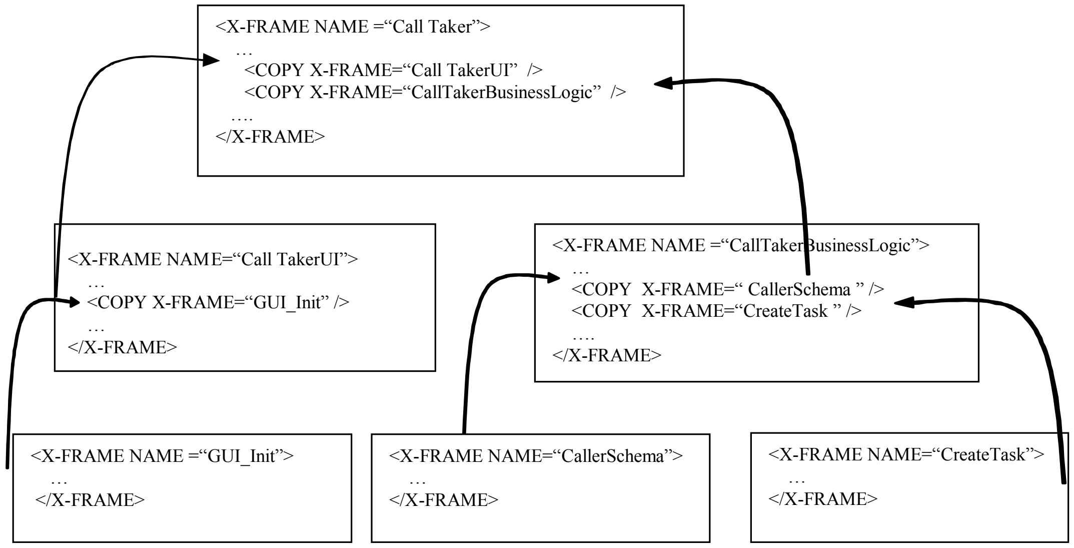

Figure 4. An example from x-frame architecture for CAD product family

5.3 XVCL: Frame Commands in XML

XVCL (stands for XML Variant Configuration Language) is an XML version of frame command language, with certain extensions for distributed applications such as CAD systems. Customization of an XML-based generic architecture, in order to produce a specific member of a product family, is done by *x-frame processor*, a tool built with the Java API for XML Parsing (JAXP). The x-frame processor traverses an x-frame hierarchy and assemblies/customizes x-frames by interpreting commands described in this section. Table 1 gives a brief summary of important frame commands and their XVCL counterparts.

Frame Command	XVCL Command	Description
frame	<X-FRAME NAME="*name*" > </X-FRAME>	Represents an x-frame.
.COPY .END-COPY	<COPY X-FRAME="*name*" > </COPY>	Allows multiple x-frames to be spliced together. When the x-frame processor encounters the COPY command, it includes a copy of specified x-frame and applies customization commands that may be (optionally) specified.
.INSERT .INSERT-BEFORE .INSERT-AFTER	<INSERT BREAK="*break-name*"> </INSERT> <INSERT-BEFORE BREAK="*break-name*"> </INSERT-BEFORE> <INSERT-AFTER BREAK="*break-name*"> </INSERT-AFTER>	This customization command inserts custom code into/before/after breakpoint *name* into the matching BREAKs within the subtree of frames rooted in the frame containing the INSERT.
.BREAK .END-BREAK	<BREAK NAME=" *break-name* " /> or <BREAK NAME=" *break-name*"> </BREAK>	Caters for unexpected new requirements. Marks a point or a block of default code in an x-frame where custom code can be inserted, default code overridden, etc.
.SELECT .WHEN .ORWHEN .OTHERWISE .END-SELECT	<SELECT OPTION="*variable*"> <OPTION VALUE="*value*"> </OPTION> ..<OTHERWISE> .. </OTHERWISE> </SELECT>	Selects none or many customization options based on the *value* of *variable*.
.REPLACE *variable* BY *value*	<SET NAME="*variable-name*" VALUE="*value*"/>	Set a value of a variable.
;*variable*!	<VAR NAME="*variable-name*"/>	Mark a location where the value of the variable can be substitute.

Table 1. Summary of important XVCL commands

We shall illustrate XVCL commands with examples from the CAD product family. Let us examine a variant involving Call Taker and Dispatcher roles. In some CAD systems, Call Taker and Dispatcher roles are played by two different people, while in other CAD systems the Call Taker and Dispatcher roles are played by one person. We use a customization option STAKEHOLDER to indicate this anticipated variant requirement. If the value of this option is "MERGED", then code for handling "Call Taker and Dispatcher roles are played by one person" will be processed. If the value of this option is "SEPARATED", then code for handling "Call Taker and Dispatcher roles played by two people" will be processed. A <SELECT> command allows us to choose a required option value (Figure 5).

```
<X-FRAME NAME=  "CallTakerUI">
<BODY>
public class OperatorUI {
    int  nState;
…. <!- - More attribute definitions here - - >
public CallTakerUI () {
….          <SET NAME="STAKEHOLDER" VALUE="SEPARATED"/>
            < SELECT OPTION=" STAKEHOLDER ">
                <OPTION VALUE ="MERGED ">
                    <!- - Some code here, Call Taker and Dispatcher roles are play by one person- - >
                </OPTION>
              <OPTION VALUE=" SEPARATED">
                  < !- - Some code here, Call Taker and Dispatcher roles are played by two people- - >
                </OPTION>
            </SELECT>
            }
            …  < !- -More method definitions here, with additional customization instructions - ->
} </BODY>
</X-FRAME >
```

Figure 5. Handling STAKEHOLDER variant with SELECT

The value of option STAKEHOLDER would be normally set to a default value (say SEPARATED). Any higher level x-frame can override the default by including the following command:

```
<SET NAME="STAKEHOLDER" VALUE="MERGED"/>
```

An x-frame can be extended with new unexpected requirements. <BREAK>s mark the points in an x-frame where unexpected changes may occur. The parent x-frame can override the default code at <BREAK> and/or insert extra code before or after the <BREAK>. The x-frame Schema in Figure 6 shows a skeleton database schema class with common parts and variation points. The x-frame CallerSchema in Figure 7 shows how the CallerSchema makes use of the x-frame Schema to produce a component that can represent and create the Caller database table. DispatcherSchema, TaskManagerSchema, FieldUserSchema, etc., can be defined in the similar way.

```
<X-FRAME NAME="Schema">
<BODY>
...
public class <VAR NAME="SchemaName"/>
{
  <!- - Common attributes here - ->
  ...
  <!- -Unexpected attributes here - ->
  <BREAK NAME="attributes"/>

  public void create(String url, String username, String
password)
  {
    try
    {
      ...
      <!- - Custom SQL statements here - ->
      <BREAK NAME="CreateBody"/>
      ...
    } catch(Exception e)
    {
      System.err.println("Problems creating table." +
e.getMessage());
    }
  }
}
 </BODY>
</X-FRAME>
```

Figure 6. x-frame Schema instrumented for change

```
<X-FRAME NAME="CallerSchema">
    <BODY>
      <SET VAR="SchemaName" VALUE="CallerSchema">
        <COPY X-FRAME="Schema">
          <INSERT BREAK="attributes">
            private String name;        // Name of caller
            private String address;     // address of caller
                ...
          </INSERT>
          <INSERT NAME="CreateBody">
            // SQL statement to create Caller table
              ...
          </INSERT>
        </COPY>
    </BODY>
</X-FRAME>
```

Figure 7. Handling unexpected variants

In the original frame language, selection of options is done by iterating over the SELECT construct in a WHILE loop. In XVCL, we introduced multi-valued variables for the purpose of selecting options. Suppose we have an x-frame to create an user interface panel with any (or none) of the five buttons: validate, save, language, view and caller info. The following SELECT element would cater for these variants:

```
<SELECT OPTION="UI_BUTTONS"/>
   <OPTION VALUE="validate">
      <!--code for validate button-->
  </OPTION>
   <OPTION VALUE="save">
      <!--code for save button-->
   </OPTION>
   <OPTION VALUE="language">
      <!--code for language button-->
   </OPTION>
   <OPTION VALUE="view">
       <!--code for view button-->
    </OPTION>
    <OPTION VALUE="callerInfo">
       <!--code for callerinfo button-->
   </OPTION>
</SELECT>
```

To create three of these five buttons in a user interface panel, say validate, save and language, we set the UI_BUTTONS variable as follows:

```
<SET NAME="UI_BUTTONS"
VALUE="validate,save,language "/>
```

6. EXTENDING FRAME COMMANDS

XVCL is backward compatible with the original frame language. So, for example, in addition to multi-valued variables for selecting options we also implemented WHILE command that is used in the original frame language for the same purpose. Being XML-based, XVCL can be easily extended by end-users to cater for specific needs of an application domain, programming language, platform or even specific project. In this section, we describe some of the extensions we found useful in CAD domain and in the *Conclusions* we discuss extensions we plan to investigate in the future work. We introduced the concept of an x-package as a logical grouping of x-frames that serve some common purpose. We can group x-frames into x-packages based on any arbitrary criteria and any given x-frame may belong to many different x-packages. In CAD domain, for example, we created x-packages for x-frames related to different stakeholders. CAD systems are component-based distributed systems implemented in Java. A typical CAD system consists of components for different stakeholders (e.g. Call Taker, Dispatcher or Task Manager). Some of the x-packages related to those stakeholders are depicted in Figure 8. The <IMPORT> command indicates the x-packages a given x-frame needs to use. For example, the SPC of Figure 8 imports all the x-packages from the CAD directory. Any subsequent <COPY> command refers to x-frames contained in imported x-packages. This feature allows us to scope the names of x-frames and resolve name conflicts. The value of X-PACKAGE attribute in <IMPORT> command can also indicate an URL address where the x-packages reside. The x-frame processor uses the information in the <IMPORT> command to search for the corresponding <X-FRAME>. This allows distribution and sharing of x-frames over the Internet.

Variable values propagate from an x-frame down to x-frames in imported x-packages, according to the same rules as we find in frame technology [1]. This is illustrated by the variable "DB_OPT" in Figure 8. In some CAD systems, database access functions are localized in one component, while in other CAD systems the database access functions are distributed among all stakeholders. The value of variable "DB_OPT" indicates whether we need a "CENTRALIZED" or "DISTRIBUTED" database. In the SPC of Figure 8, we make certain choices based on the value of the variable "DB_OPT". The value of "DB_OPT" propagates down the x-frame hierarchy to trigger similar changes in all the affected x-frames.

```
<SPC NAME=“CAD">
            <!-- SPC for producing CADsystem in which call taker and dispatcher roles merged with distributed database -->
       <DECLARATION>
              <IMPORT  X-PACKAGE=“CAD.*"/>  <!-- import all the x-packages from CAD directory-->
       </DECLARATION>
       <BODY>
            <SET NAME=“DB_OPT" VALUE=“DISTRIBUTED"/>
            <SET NAME=“STAKEHOLDER" VALUE=“MERGED"/>
            <SELECT OPTION=“DB_OPT">
              <OPTION VALUE=“CENTRALISED">
             <COPY X-PACKAGE=“DATABASE" X-FRAME=“CentralizedDatabase" OUTDIR=“database" OUTFILE=“DBAccess"/>
              </OPTION>
            </SELECT>
            <COPY X-PACKAGE=“CALLTAKER" X-FRAME=“CallTaker"  OUTDIR=“calltaker" OUTFILE=“CallTaker"/>
            <COPY X-PACKAGE=“DISPATCHER" X-FRAME=“Dispatcher" OUTDIR=“dispatcher" OUTFILE=“Dispatcher"/>
         <COPY X-PACKAGE=“TASKMGR" X-FRAME=“TaskManager" OUTDIR=“taskManger" OUTFILE=“TaskManager"/>
            <COPY X-PACKAGE=“FIELD_USER" X-FRAME=“FieldUser" OUTDIR=“fielduser" OUTFILE=“FieldUser"/>
       </BODY>
</SPC>
```

Figure 8. Examples of x-packages in CAD x-frame architecture

The other extension that we implemented in the XVCL relates to processing of frames written in multiple languages. Attributes LANGUAGE, OUTDIR and OUTFILE cater for language-specific processing, for example:

<X-FRAME NAME=“DispatcherUI” LANGUAGE=“JAVA” OUTDIR=“Dispatcher” OUTFILE=“DispatcherUI.java”>

The LANGUAGE attribute identifies the language the x-frame’s contents is written in. Attributes OUTDIR and OUTFILE tell the x-frame processor how to format and organize the output from the x-frame processor. For example, in Java, each public class resides in a file and related classes are organized into packages, which in turn become directories. We can cater for such language-specific requirements with the above attributes.

7. THE X-FRAME PROCESSOR

The x-frame processor traverses the x-frame hierarchy starting from the root. For each visited x-frame, the processor interprets specified customization commands and emits the output accordingly.

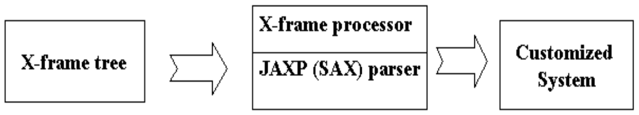

Figure 9. The x-frame processor

We implemented the x-frame processor on top of the public domain XML parser. Among many free XML parsers available (from IBM, Microsoft, Sun and Apache), we chose JAXP from Sun because of its simplicity and lightweight. XML parsers provide both SAX and DOM parsers. For speed and efficiency, we built x-frame processor using validating SAX parser. Implementation of XVCL processor with JAXP is simple. The XML parser eliminates many tedious tasks and the programmer can concentrate on the essential task of writing document handlers for various commands.

8. CONCLUSIONS

Macro processors have been applied to both procedural and Object-Oriented languages [5], however they have never been accepted as an effective technique to handle variants in product lines [12]. Frame technology is a form of advanced macro system [1]. It is therefore both intriguing and worth effort to investigate the principles that have made frame technology so successful. This was the main motivation for undertaking the work described in this paper and in our previous projects.

Frame technology has been developed to handle large COBOL-based business software product families. We wished to investigate how the principle of frame approach can be applied to support product families in other application domains, in particular to build distributed component-based systems written in Object-Oriented languages. As Fusion™ is tightly coupled with COBOL, we implemented our own tools based on frame concepts using the XML technology. In our solution, a generic architecture for a product family is a hierarchy of XML documents. Each such document contains a reusable program fragment instrumented for change with XML tags. We used a public domain XML parsing framework JAXP to process documents in order to produce a custom member of a product family.

Our solution is simple and extensible. In particular, it is easy to extend frames to address needs of a specific application domain, implementation language and platform. XML standards and widely available XML parsers (such as JAXP) offer ample room to implement many other tools (such as documentation tools, browsers, smart editors, program transformation and reverse engineering tools) that can further support the product family. With permission of Netron™, we intend to make our solution available to public and encourage wide investigation of frame concepts in order to accumulate experiences from different types of software projects. This should also encourage comparative studies of emerging approaches such as aspect-oriented programming, meta-programming, intentional programming and multi dimensional hyperspaces to evaluate their effectiveness in supporting different types of variants in product families.

We plan to initiate such studies and further investigate frame principles using our XML tool. In our current project, we are experimenting with advanced forms of separation of concerns [17] in framed architectures. In particular, we are trying to explicitly define slices of an x-frame architecture related to:

- a subsystem (e.g., order processing, billing, report generation) or component (e.g., Call Taker or Dispatcher in CAD systems)
- security, communication, load-balancing algorithms, process scheduling,
- user interface, business logic, database,
- component interface definitions,
- middleware, and many other concerns that matter in a given product family.

We are extending XVCL to provide better support for separation of concerns. In the future work, we shall further explore application of frame concepts to distributed component-based product families and mission-critical systems such as CAD. We are also interested in how frame approach blends with component-based systems, component platforms and design methods for distributed applications in Object-Oriented languages such as Java. We designed a generic CAD architecture by modular decomposition of a problem and solution space into components, using principles of abstraction, separation of concerns, information hiding and design patterns. We used Java language and component platform features that support such design. At the same time, we applied concepts of frames to address variants that cannot be localized to a small group of modules. We believe blending component-based approach with frame concepts is a promising way to tackle difficult issues in design of flexible software systems.

9. ACKNOWLEDGEMENTS

We acknowledge with thanks contributions of Paul Bassett to the work described in this paper. Paul is a collaborator in our projects and it was his idea to express frame language and processor in the XML environment. He and Karl Wagner of Netron, Inc. offered their invaluable advice throughout the project to ensure that we correctly interpret frame concepts in our implementation. We are indebted to Netron Inc. for letting us use their product Fusion in our projects and personally to Paul Bassett for his invaluable contributions to this and our other projects. The work on the XML implementation of the frame concepts was intiated by NUS student Wong Tak Wang [19] and continued by Soe Myat Swe. This project was supported by research grant NSTB/172/4/5-9V1 funded under the Singapore-Ontario Joint Research Programme by the Singapore National Science and Technology Board and Canadian Ministry of Energy, Science and Technology, and by NUS Research Grant R-252-000-066.

10. REFERENCES

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

[2] Batory, D., Lofaso, B. and Smaragdakis, Y. JST: Tools for Implementing Domain-Specific Languages. In 5th Int. Conf. on Software Reuse, Victoria, BC, Canada, 1998, pp. 143-153

[3] Brown, A. and Wallnau, K. The Current State of CBSE, IEEE Software, September/October, 1998, pp. 37-46

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

[5] Chiba, S. Macro processing in object-oriented languages, in proceedings of TOOLS'98, Editor(s): Mingins, C., Meyer, B. Inst. of Inf. Sci. & Electron., Tsukuba Univ., (Ibaraki, Japan, 1998), pp. 113-126

[6] Czarnecki, K. and Eisenecker, U. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models, Addison-Wesley, 2000

[7] DeBaud, J.M. and Schmid, K. A Systematic Approach to Derive the Scope of Software Product Lines. Int. Conf. on Software Engineering, ICSE'99 (Los Angeles, May 1999), pp. 34-43

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

[9] Goguen, J. ,Parameterized programming, IEEE Transactions on Software Engineering, SE-10, No. 5, pp. 528-543. 1994.

[10] Jarzabek, S. and Knauber, P. Synergy between Component based and Generative Approaches, in proceedings of ESEC/FSE'99 Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, (Toulouse, France, September 1999), Lecture Notes in Computer Science No. 1687, pp. 429-445

[11] Johnson, R. and Foote, B. Designing reusable classes, Journal of Object-Oriented Programming, 1988, pp. 22-35.

[12] Karhinen, A., Ran, A. and Tallgren, T. Configuring designs for reuse, in Proceedings of ICSE'97 (Boston, MA, 1997). pp. 701-710.

[13] Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. Aspect-Oriented Programming. in Proceeding of ECOOP'97, (Finland, 1997), Springer-Verlag LNCS 1241.

[14] Parnas, D.L. On the design of program families. IEEE Transactions on Software Engineering, SE-2, 1976, pp. 1-9.

[15] Sommerville, I. and Dean, G. PCL: A language for modeling evolving system architectures, Software Engineering Journal, 1996, pp. 111-121.

[16] SUN Java Technology and XML. Sun Microsystems, Inc. Retrieved July 23, 2000 from the World Wide Web: http://java.sun.com/xml/

[17] Tarr, P., Ossher, H., Harrison, W. and Sutton, S. N Degrees of Separation: Multi-Dimensional Separation of Concerns, Int. Conference on Software Engineering, ICSE'99, (Los Angeles, 1999), pp. 107-119

[18] W3C 1998. Extensible Markup Language (XML) 1.0. REC-xml-19980210, W3C. Retrieved July 23, 2000 from the World Wide Web: http://www.w3.org/TR/REC-xml

[19] Wong, T. W. Methods and tools for a generic Computer Aided Dispatch System, M. Sc. Thesis, School of Computing, National University of Singapore, September 2000