A Review of Frame Technology

Chris Holmes
Andy Evans
Real-Time Systems Group
Department of Computer Science
University of York
Heslington
York YO10 5DD
UK

28 November, 2003

E-mail: chris.holmes@cs.york.ac.uk
andye@cs.york.ac.uk

Abstract

Frame Technology was introduced during the early 1980's as an approach to providing significant levels of improvement in software productivity through adaptive reuse. However, despite the claims of order of magnitude improvements and reuse levels of 90%, this technology does not appear to have acquired a corresponding level of popularity, appearing to loose out to the more recent technologies of *generative programming* and *aspect oriented programming*. Our interest frame technology stems from our work on the meta-modelling of feature models to support the expression of product line architectures in the UML using a meta-modelling research tool. Whist the tool provides direct support for *templates*, we sought an approach that would approximate to customisable templates, allowing us to generate specific *configurations* in response to user *choices* by severing unused features. Frame Technology appears to offer a potential solution and we present a summary of our investigation of this technology. Finally, we propose future directions of study based around a meta-modelling approach.

1 Introduction

We are investigating the specification of a UML metamodel of *features* [2] following a precise approach to meta-modelling in the form advocated by Clark et al. [1]. We take our definition of the concept of a *feature* from Czarnecki and Eisenecker [24]:

> *'A property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances.'*

It is our intention that the metamodel should be suitable for future publication in the form of a clear and unambiguous *profile* that could be used to extend the UML to provide support for feature modelling and the Product Line Architecture (PLA). A UML profile is essentially a horizontal or vertical extension to the core language, and may be thought of as roughly analogous to the specialised needs annexes of the Ada programming language [3]. In order to provide a clear semantics of the model elements comprising the proposed feature modelling profile, we intend that the metamodel should be checkable by a tool. Work on the specification of a metamodel of features commenced using the tool MMT [4]. This tool allows the definition of a metamodel in the form of Java-like programs, the user is then able to exercise the model by drawing snapshots (instances of the metamodel) to verify correctness. Further work was undertaken with the USE tool [5]; this works in a similar fashion to MMT, although it is limited to the importing of a single model. Work is now being migrated to the new tool XMT [6], which supports both graphical and text-based (meta) model specification and validation, and an incremental approach to the construction of models. Our view of feature modelling is derived (primarily) from Kang et al. [7] and [12]. We define a number of primitive components to allow the construction and relating of features in the form of well-formed and valid feature diagrams. A family member (configuration) is simply an instance[1] of the feature diagram conforming to certain configuration rules (constraints) expressed in the feature model, and is the result of user choices, e.g.

$$FeatureModel \times Choices \Rightarrow ConfigurationModel$$

We wished to ensure that the configuration model did not carry with it a copy of unused features, hence we required a more adaptable approach to the generation of the configuration model than a regular template, since this would maintain the structure (although one could possibly mark unused features). Furthermore, we wished to model the change of focus

[1] In our view these reside at the same meta-layer – this also alleviates the application of OCL constraints.

of feature semantics in the configuration view, since, at this stage, optionality (in terms of a system model) is redundant, and feature relationships (e.g. in terms of hyponyms and meronyms) are now more relevant. Hence, we required some form of template that would support the instantiation of model fragments (see Figure 1-1).

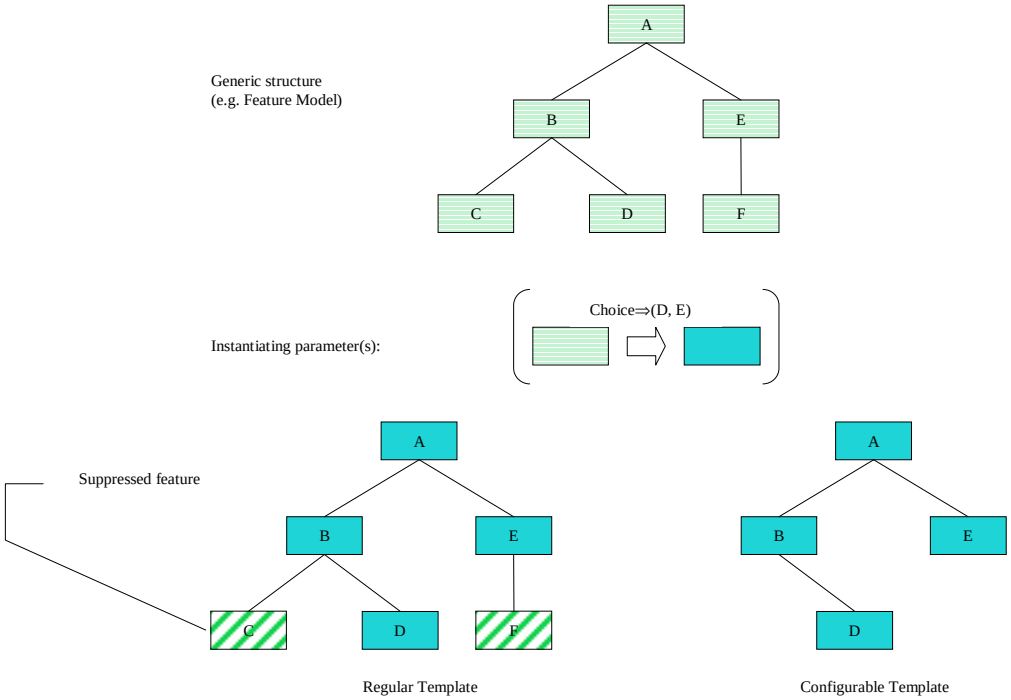

Figure 1-1 - Templates and Configurable Templates

The notion of constraining, adapting and instantiating a structure based on user-directed choices is the guiding theme of Frame Technology [9], hence we have investigated the emulation of a configurable template via frames.

The remainder of this paper is structured as follows:

- Section 2 provides an introduction to Frame Technology
- Section 3 describes a number of approaches to frame processing
- Section 4 discusses the application of frame technology
- Section 5 describes related work
- Section 6 provides our conclusions

2 Frame Technology

Paul Bassett first develope the ideas underpinning Frame Technology during the early 1980's as a tool to support the automated fabrication of software systems from a hierarchy of adaptable components, although it wasn't until 1997 that this information was published in the form of a book. Bassett reports that frame technology practitioners have achieved high levels of software reuse (>90%) and significant productivity advantages over their competitors. Bassett quotes a study showing that companies employing Frame Technology (marketed by Netron Inc. as part of the Fusion™ toolset) within the Information Management System (IMS) domain achieved development costs and time-to-market of 84% and 70% lower than the industry average [9]. Although frames are language-independent, the primary area of uptake discussed in [9] relates to large COBOL programs (from 4.5KLOC to 9.3MLOC). Netron Inc. provides support for Frame Technology via the Fusion™ toolset, although this is known to be tightly coupled with COBOL [13]. We also note that there is anecdotal evidence of the use of Frame Technology by Siemens Nixdorf Information Systems to support the automatic generation of documentation from adaptable software components. In [37] Tilbrook and Crook describe the use of parameterised prototype files used to configure systems for various target environments using a simple frame-like approach.

Frame Technology epitomises the *same as except* style of reuse; when writing software systems, we often start with something similar that has been developed in the past and then begin adapting it to suit our current needs. Frames make this approach explicit by the identification of variation points [10] in the framed component (via frame commands). The user (re-user) provides appropriate parameters and/or values to specify the desired variation, typically via a single specification frame (SPC), and a frame processor then generates the specified component(s) by traversing the network (lattice) of frames and processing the frame commands. The set of frame commands is relatively small and provides a

Turing-complete language[2], naturally, all frame commands are executed at generation time. In this context, frames may be viewed as meta-programs[3]. The frame processor applies transformations to the framed component, based on user choices, and generates a product; hence, a frame's interface may support the generation of many unique products. The ability to generate custom products on demand is imperative as it helps to address the scalability problem identified by Biggerstaff that affects concrete component reuse libraries [11]. Conventional reuse libraries tend to grow in a combinatorial fashion as new features are added, resulting in a large number of similar components. This adds to the configuration management effort, and, more importantly, may make selection of the appropriate component from its many similar counterparts much more difficult for the user.

2.1 Frames

A frame may be viewed as a parameterised wrapper around a *component*. Frame Technology is language and domain independent, the frame comprising only commands for the frame processor and text (the component). Hence, the framed component may be source code, test data, documentation, web pages, etc. The frame exports an interface identifying variation points, and may, itself, customise subordinate frames by their variation points. Variation points allow the user to modify aspects of the product to be generated. By specifying appropriate parameters, the user is able to add, delete, copy, and adapt the framed component; the reused frame may itself customise subordinate frames; in this way the construction of sub-components is delegated to sub-frames. Furthermore, multiple superordinate frames may reuse common sub-frames in differing ways; hence the frame hierarchy resembles a lattice rather than a tree.

2.2 The Frame Command Language

The frame command language is a small, relatively simple language to support the customisation of framed components in accordance with user-specified parameters. The frame command language as described by Bassett in [9] supports the following features:

- Overriding
- Extension
- Substitution
- Selection
- Iteration

Each of the above is described in outline below, the reader is referred to [9] for a more detailed description.

2.2.1 Overriding

A frame may allow clients to override certain aspects via the *Break* command. This command identifies a block of text with a name that may be accessed by clients such that the named block of text may be omitted, copied, or extended with additional text (see below).

2.2.2 Extension

In conjunction with the *Break* command (above), client frames may copy (import) named sections of text and (optionally) append/prepend or replace with additional text, e.g. the extending of a `Stack` frame with support for concurrency, or the addition of a `Depth` function.

2.2.3 Substitution

Key words in the framed document may be parameterised such that client frames may substitute new values at generation time, e.g. the name of a class may be generated as a result of the context in which it is processed; `Protected_Stack` being substituted for `Stack`.

2.2.4 Selection

A frame may contain a number of mutually exclusive alternatives, e.g. the dimensional systems CGS[4] or SI Units, or a number of combinable options, e.g. a number of attributes and/or operations on a class. The client frame may make the appropriate choice by providing the appropriate parameter, or accepting the default selection. Selection is provided by the *Select* command (equivalent to a *Case* statement).

[2] Supporting *sequence*, *selection*, and *iteration*.
[3] A program that generates another program.
[4] Centimetres, Grams, Seconds.

2.2.5 Iteration

A frame may contain a section that may be iterated over to generate multiple (different) instances. As an example, a frame may contain a collection class in the form of template code, by iterating over this template code with the appropriate type information we could emulate some aspects of the strong compile time checking of (e.g. Ada) in less strongly typed languages (e.g. Python).

2.3 Putting It All Together

Frames comprise a sequence of commands to the frame processor and textual elements comprising the framed component, the semantics of the framed component have no bearing on the behaviour of the frame processor; it is simply guided by the frame commands it encounters. As alluded to above, frames are typically unlikely to be stand-alone, any sizeable system would have many frames organised in the form of a lattice (hierarchical), with context-dependent frames in the higher layers, and context-independent frames in the lower layers. At the top of the hierarchy is the specification frame (SPC), this frame is the most context-dependent frame and describes how the system should be built; one would expect an instance of an SPC frame for each family member of the PLA. The SPC frame may comprise framed elements, or it may simply parameterise (and invoke) subordinate frames. The frame processor proceeds in a depth then breadth fashion through the hierarchy as the system is assembled (see Figure 2-1 and Figure 2-2).

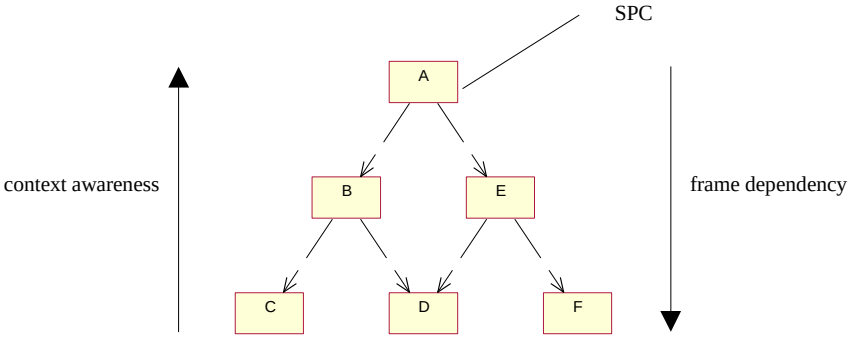

Figure 2-1 – Example Frame Hierarchy

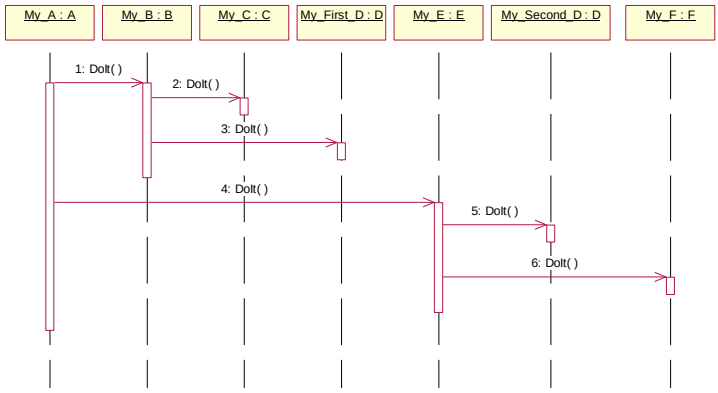

Figure 2-2 – Sequence of Execution

It is interesting to contrast the view taken by the frame-based approach against the Liskov Substitution Principle (LSP) that is generally applied to OO-based designs [14] and [15]. Under the guidelines of the LSP, it should be possible to substitute a subclass for any of its superclasses; hence reuse specialisation provides an extension to the attributes and operations of a class[5]. However, as Guthrey points out in his critique of the inherit to reuse paradigm of the late 1980's, a subclass may provide many more attributes and operations than may be necessary in a given context [16]. The frame commands, on the other hand, allow the adaptation of classes, such that attributes and operations may be suppressed and added by a frame. This isn't actually in direct contravention of the LSP, since we're operating in the generation time domain, although there is clearly scope for a pathological frame to generate non-conforming classes. Another reason for the 'same-as-except' style of reuse stems from the need to obviate component incompatibilities when composing new classes from existing classes. For example, the composition of a class VSTOL from classes Aircraft and Helicopter (see Figure 2-3) leads to a conflicting view of the Maximum Take-Off Weight (TOW). In the context of class Aircraft, this

[5] We acknowledge that there are other forms of inheritance, e.g. implementation, interface, private, etc.

relates to conventional wing borne flight, whereas, in the context of class Helicopter, this relates to the hover; but what does it mean in the context of the derived class VSTOL? One might view this style of reuse as being, on the one hand pragmatic, and, on the other, a mechanism to avoid (or defer) refactoring of the model.

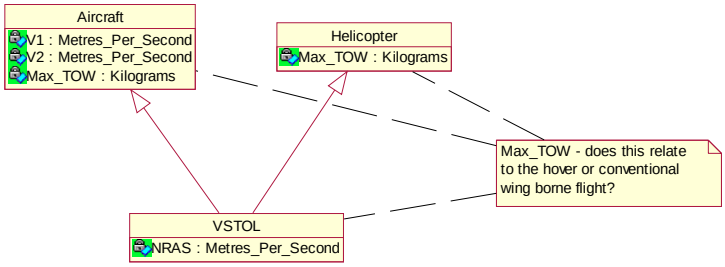

Figure 2-3 – Example of Conflicting Attributes

Another interesting aspect of the frame-based paradigm is the unification of composition and generalisation relationships, the argument being that if class X *has-a* class Y and this relationship persists for the lifetime of the system execution, then the guidance is to merge both classes X and Y [9]. Whilst this approach may, at first, seem undesirable; after all we make regular use of design patterns such as the Whole-Part pattern [17], it may be viewed as simply using the same Whole-Part patter, but at frame design time rather than code design time. Furthermore, Buschmann et al. describe the intent of the Whole-Part pattern as being the aggregation of components into a semantic unit, following Bassett's approach, the semantic unit is generated as a single structural unit. The merging of composition and generalisation relationships, combined with support for the suppression of attributes and/or operations also highlights the common use of a variation of multiple inheritance; again, this occurs at generation time, hence, the resulting products may well be generated devoid of inheritance hierarchies.

3 Supporting Frame Technology

It is reported that Frame Technology has been applied with great success within the IMS domain and, more specifically, to systems written in COBOL. However, the 'closed' nature of the toolset as a result of its tight coupling with COBOL [13] would appear to hinder the more widespread adoption of frames. It would appear undesirable for the frame processor and the Frame Command Language (FCL) to have any knowledge of, or make any assumptions about the target domain and/or language. One candidate technology that may provide such a neutral environment is the eXtensible Markup Language (XML), augmented by the eXtensible Stylesheet Language for Transformations (XSLT). XML may be used to store and manage data in the form of *documents*; we may also stipulate certain well-formedness rules on the structure of such documents. As part of our work, motivated by a need to simulate variant templates in a UML metamodel, we are investigating the application of the XML and XSLT technologies to specify and process frames. We chose XML and XSLT over (e.g.) lex and yacc [18] because of the ease of availability of editing, browsing and processing tools (e.g. Emacs[6], Netscape[7], and Xalan[8]), and because XML and XSLT provide an homogeneous solution. However, the lack of expressive power of the DTD provides a somewhat less capable vehicle for the definition of the syntax of the FCL, although adoption of the XML schema should provide support for a more rigorous definition.

In this section we provide a brief introduction to XML and XSLT by way of a simple example, we then describe the approach we have taken to the development and implementation of a simple frame processor using these technologies.

3.1.1 Overview of XML

The XML provides a toolkit for storing and managing data [19]. The core of an XML document is structured in the form of a number of user-definable elements following a user-definable structure (the content model). The XML document must correspond to certain syntactic rules; e.g. enclosed within a root element, tags must be balanced and enclosed within angled brackets of the form `<myTag>`, i.e.

```
<myTag>
    <!– child elements -->
</myTag>

<myTag/>  <!– an empty element -->
```

[6] See http://ww.gnu.org/software/emacs for details.
[7] See http://www.netscape.com for details.
[8] See http://xml.apache.org/xalan-j/index html for details.

Hence, all XML document must conform to certain rules. An example XML document is shown in Figure 3-1.

```
<messages>
    <message id=”msg-1” urgency=”urgent”>
        <header>
            <sender name=”Ground Control” address=”gctl@earth.com”/>
            <receiver name=”Mjr. Tom” address=”tom@spaced_out.com”/>
            <received date=”01-Apr-00”/>
            <title>
                Health Check
            </title>
        </header>
        <body>
            You’re circuit’s <emphasis>dead</empahsis>,
            there’s something wrong.
        </body>
    </message>
    …
</messages>
```

Figure 3-1 – Example XML Document

An XML document satisfying the basic syntax rules (freeform XML) is said to be *well-formed*. However, we can define custom markup languages by providing a more rigorous definition of a document via a Document Type Definition (DTD). In the case of the example in Figure 3-1, we might wish to mandate that the node `message` must contain a node of type `header` and (optionally) `body`, and have attributes `id` and `urgency` (see Figure 3-2). An XML document conforming to the specification in the DTD is said to be *valid*.

```
<!ELEMENT message (header, body?)>
<!ATTLIST message
          id ID #REQUIRED
          urgency (urgent | firstClass | secondClass) “urgent”>
<!ELEMENT header …>
<!ELEMENT body …>
```

Figure 3-2 – Example DTD

The document content model may be composed from a number of lower-level reusable modules wrapped-up into a DTD file. The document content model may then be referenced from within documents via a DTD declaration to request that a validating XML parser check that the document conforms to its defined content model (see Figure 3-3). It is also possible to define the markup language directly within the XML document, although this is probably only appropriate for small one-off models.

```
<?xml version=”1.0”?>
<!DOCTYPE messages SYSTEM “messages.dtd”>

<messages>
   …
</messages>
```

Figure 3-3 – Using a DTD

3.1.2 Overview of XSLT

The XSLT is a relatively simple, structured high-level language to support the processing, and, more importantly, transformation of XML documents. XSLT conforms to a very similar syntax as XML and together they provide an homogeneous solution to data representation, exchange and processing. There are alternative approaches such as the Document Object Model (DOM) and Simple API for XML (SAX), however these rely on the support of a conventional programming language, such as Java or Python with the appropriate libraries. The user must provide extensions in the form of (e.g.) event handlers plugged into a SAX implementation to provide the required interpretations of document instances [36]. XSLT supports the transformation of XML documents to three target types: HTML, XML, and text. Hence, we can write document transformations to (e.g.) render data for a web browser (see Figure 3-4), merge documents with common or differing document models (see [20] for various common transformations), or generate a custom report as text; and this *text* could be code (fragments).

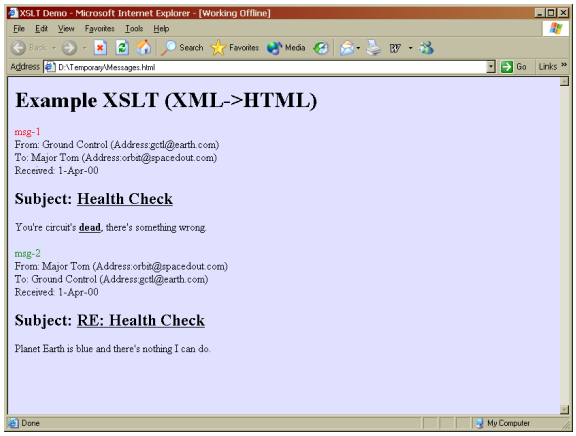

Figure 3-4 – Example of XML rendered as HTML

Programming under XSLT is generally based on a combination of pattern matching and imperative programming; e.g. the code fragment in Figure 3-5 was used to render the XML document `messages` as the HTML shown in Figure 3-4. XSLT transformations may be applied to XML documents in a couple of ways:

- Implicitly, via an XSLT aware browser (e.g. IE6) by embedding a reference to the stylesheet in the source XML document, the browser will then apply the transformations and render the resulting HTML.
- Explicitly, via a standalone XSLT processor such as Xalan.

In the case of the former, the transformation is being bound to the source document, this is compact, but at the cost of flexibility; we may wish to perform any of a number of transformations on a particular document. In the case of the latter the document and transformation are kept separate, thereby affording the flexibility of rendering the same source document in a variety of styles, although at the cost of an explicit transformation stage.

It should be noted that XSLT does have a few restrictions, such as the redirection of I/O streams, however popular XSLT processors provide support via extensions[9].

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

    <!-- main -->
    <xsl:template match="/">
        <html>
            <head>
                <title>
                    <xsl:value-of select="messages/@applicationName"/>
                </title>
                <h1>
                    <xsl:text>Example XSLT (XML->HTML)</xsl:text>
                </h1>
            </head>
            <body bgcolor="#e0e0ff">
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="message">
        <xsl:variable name="status">
            <xsl:value-of select="@urgency"/>
        </xsl:variable>
        <xsl:variable name="value">
            <xsl:value-of select="@id"/>
        </xsl:variable>
        <xsl:call-template name="banner">
            <xsl:with-param name="priority"
                            select="$status"/>
            <xsl:with-param name="item"
                            select="$value"/>
        </xsl:call-template>
        <xsl:apply-templates/>
        <br/>
    </xsl:template>
```

```
<xsl:template name="banner">
        <xsl:param name="priority"/>
        <xsl:param name="item"/>
        <xsl:choose>
            <xsl:when test="$priority='urgent'">
                <font color="red">
                    <xsl:value-of select="$item"/>
                </font>
            </xsl:when>
            <xsl:otherwise>
                <font color="green">
                    <xsl:value-of select="$item"/>
                </font>
            </xsl:otherwise>
        </xsl:choose>
        <br/>
    </xsl:template>

    <xsl:template match="header">
        <xsl:apply-templates/>
    </xsl:template>

<xsl:template match="sender">
        <xsl:text>From: </xsl:text>
        <xsl:call-template name="nameAndAddress"/>
        <br/>
    </xsl:template>

    <xsl:template name="nameAndAddress">
        <xsl:value-of select="@name"/>
        <xsl:text> (Address:</xsl:text>
        <xsl:value-of select="@address"/>
        <xsl:text>)</xsl:text>
    </xsl:template>

    <snip>
    </snip>

</xsl:stylesheet>
```

Figure 3-5 – XSLT code fragment

The example code in Figure 3-5 shows both the pattern matching form of template (e.g. `template match="message"`) and the imperative form of template invocation (e.g. `template name="banner"`). Processing commences at the document root ("/") and the templates are triggered by the matching of certain elements in

[9] Xalan provides the `redirect` command.

the source (XML) document (e.g. element `message`). The code `<xsl:apply-templates/>` instructs the XSLT processor to match any template to elements in the current context, hence the call in template header would try to find templates (elided) to handle the child elements `sender`, `receiver`, `received`, and `title`.

3.1.3 Opening Frame Technology

By this stage the reader should have gained a general understanding of both Frame Technology and the capabilities of the XML and XSLT technologies. Although there are a number of frame processors available in the form of both free and commercial tools, we are not aware of any that are open to customisation of the underlying FCL. We believe that the use of XML and XSLT could support the development of an open and extensible frame processor to satisfy our requirement for support of variant templates for the meta-modelling tool XMT. Furthermore, whilst we are aware of other approaches to frame processing using XML [13] and [21], we are not aware of any using XSLT to implement the frame processing engine. We now describe our approach to the definition of a simple open and extensible frame processing environment using XML and XSLT.

3.1.3.1 Key Components

There are essentially three components required by a frame processing environment: customisations, frames and transformations. We describe customisations and frames in the form of XML documents and the transformations in the form on XSLT *scripts*. We provide some degree of integrity checking for the customisation and frame documents by way of application-specific DTD files. We describe a number of schemata in the form of DTD module files that are imported by the higher level module files and, ultimately, DTDs; we are then able to validate our XML documents against the relevant DTD(s); Figure 3-6 shows the structure of our core component modules.

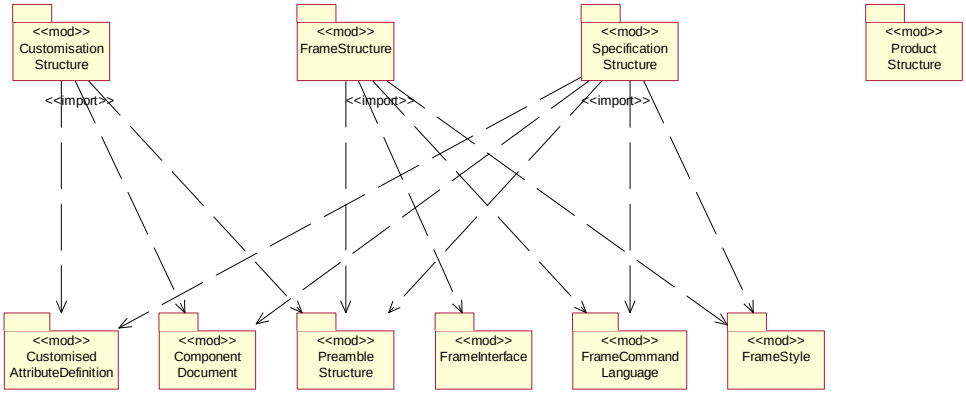

Figure 3-6 – Core Document Schemata

We describe common structures (document models) for specific artefacts, such as a Frame, however these are expected to be extended by the importing application domain via the completion of the document schema description from a predefined hotspot; we currently use the element `<line>` as the extension point. Hence, frame documents must conform to both domain-dependent and domain-independent constraints, such constraints being described by the composition of the relevant document model components into a single frame-specific DTD.

3.1.3.1.1 Frame Command Language

Bassett defines the following primary fame commands [9]:

- `Break`
- `Copy/Insert`
- `Replace`
- `Select`
- `While`

Currently, we provide only support for break-out points, selection, and comments. We describe the generic structure of a frame; essentially, it consists of a preamble and a body section. The preamble section is used to provide documentation about the frame; its name, description, the DTD to be used to validate the specification to be generated, etc. The body section comprises lines (the point of domain-specific extension) and frame commands, and must adhere to the grammar shown in Figure 3-7.

```
<!ELEMENT contents (line* | break* | case* | comment*)*>

<!ELEMENT break (line* | break* | case* | comment*)*>
<!ATTLIST break name ID #REQUIRED>

<!ELEMENT case (when+, otherwise)>
<!ATTLIST case name CDATA #REQUIRED>
<!ELEMENT when (line* | break* | case* | comment*)*>
<!ATTLIST when test CDATA #REQUIRED>
<!ELEMENT otherwise (line* | break* | case* | comment*)*>

<!ELEMENT comment (#PCDATA)>

<!-- <!ELEMENT line (isFrameSpecific)> →
```

Figure 3-7 – FCL Grammar

The FCL grammar states that a `contents` element may contain any number (including zero) of `line`, `break`, `case` or `comment` elements in any order. The `break` element, as one would expect, delineates a section of the component and is identified by a unique `name` attribute, the `case` element follows the syntax of the XSLT `choose` element, the `comment` element simply contains parsed character data, and the `line` element is the point of frame-specific extension. The module `FrameStructure` merges the FCL definition with a number of other components (see Figure 3-6) and provides the generic structure for frame documents. We then merge the generic frame structure with an application-specific definition of the `<line>` element (the point of extension) to provide an application-specific DTD, e.g. see Figure 3-8.

```
<!-- Import the generic frame structure MOD file.          -->
<!ENTITY % genericStructure SYSTEM
   "/home/chris/Frame_Technology/Resources/FrameStructure.mod">
%genericStructure;

<!-- Import Humpty Dumpty related mark-up.                 -->
<!-- This is where we provide the application-specific     -->
<!-- completion of the 'line' tag that is exported by the  -->
<!-- FrameCommandLanguageModule as a point of extension.   -->
<!ENTITY % humptyDumptyStructure SYSTEM "HumptyDumpty.mod">
%humptyDumptyStructure;
```

```
<!ELEMENT line (#PCDATA | monarch | lead)*>
<!ELEMENT monarch EMPTY>
<!ATTLIST monarch name (King   |
                        Queen  |
                        Prince |
                        Princess) "King">
<!ELEMENT lead EMPTY>
<!ATTLIST lead name (HumptyDumpty |
                     ContraryMary |
                     TopsyAndTim  ) "HumptyDumpty">
```

Figure 3-8 – Definition of Application-Specific Schema

We are now able to write our frame in terms of the generic document model (including the FCL) and application-specific constraints, all specified via a single DTD that has been composed from a library of reusable schema fragments (see Figure 3-9).

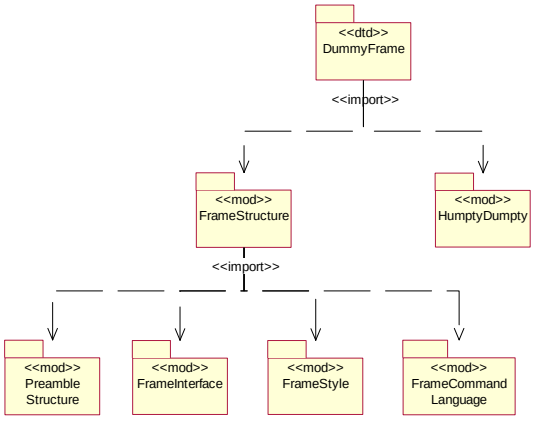

Figure 3-9 – Composition of Document Schema

Figure 3-10 shows an example frame component following the Humpty Dumpty theme demonstrated by Bassett [9] conforming to the composed document schema illustrated above. It will be observed that this frame comprises a variety of components that could be generated by appropriate customisations.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE frame SYSTEM "DummyFrame.dtd">

<frame>
  <preamble>
    <file name="DummyFrame.xml"/>
    <description>This is a test file.</description>
  </preamble>
  <frameStyle type="product"/>
  <body>
    <contents>
      <line><lead/> sat on a wall</line>
      <line><lead/> had a great fall</line>
      <line>All the <monarch/>'s horses and all the <monarch/>'s men</line>
      <line>Couldn't put <lead/> together again.</line>
      <case name="ending">
        <when test="happyEverAfter">
          <line>But it's OK, <lead/> recovered.</line>
        </when>
        <when test="irony">
          <line>Unfortunately, <lead/> cracked-up.</line>
        </when>
        <otherwise>
          <line>Poor <lead/> died.</line>
        </otherwise>
      </case>
      <case name="epitaph">
        <when test="burialAtSea">
          <line><lead/> was commended to the sea!</line>
        </when>
        <when test="burial">
          <line><lead/> is now six feet under!</line>
        </when>
        <otherwise>
          <line>Sadly, <lead/> was cremated!</line>
        </otherwise>
      </case>
    </contents>
  </body>
</frame>
```

Figure 3-10 – Example Frame

3.1.3.2 Document Transformations

We currently provide two XSLT scripts to support the generation of a *product* from a frame (see Figure 3-11), a third transformation stage (to be developed) transforms the product description into the intended component – i.e. a code fragment or, alternatively, another customisation. In the former case we transform to text (e.g. Ada code), and in the latter to XML, enforcing the customisation schema (only the former case is shown in Figure 3-11).

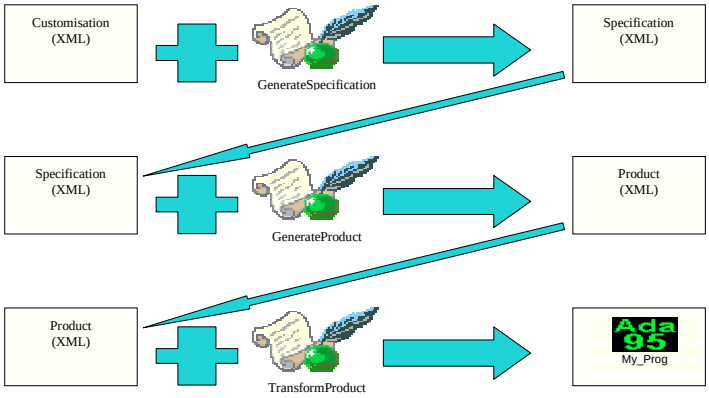

Figure 3-11 – Transformation Scripts

In our current approach we choose to distinguish between customisations and frames, although this distinction has been made primarily to support debugging of the XSLT transformations[10]. Hence, we view the frame as being a template that must be parameterised by a customisation (a null customisation would lead to the generation of a frame's default component). This is really an unnecessary complexity, and we would propose to revert to the more classical (and symmetric) view that frames adapt frames (directly). We also view the frame as being a black box, i.e. the customisation knows about the frame(s) to be used, but the frame has no knowledge of its clients. The customisation files are structured in the form shown in Figure 3-12.

[10] At the time of writing there are no freely available XSLT debuggers.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE frameCustomisation SYSTEM "DummyFrameCustomisation.dtd">

<frameCustomisation>
  <preamble>
    <file name="DummyFrameCustomisation.xml"/>
    <description>This is a customisation test file.</description>
  </preamble>
  <frame name="DummyFrame.xml">
    <specification name="HumptyDumptySpecification.xml"
                   dtd= "HumptyDumptySpecification.dtd"/>
    <component     name="/home/chris/Frame_Technology/Test/HumptyDumpty.xml"
                   dtd= "/home/chris/Frame_Technology/Test/HumptyDumpty.dtd"/>
    <customisation tagName="lead"
                   attributeName="name"
                   value="HumptyDumpty"/>
    <customisation tagName="case"
                   attributeName="ending"
                   value="gruesome"/>
    <customisation tagName="case"
                   attributeName="epitaph"
                   value="burialAtSea"/>
  </frame>
  <frame name="SoundEffects.xml">
    <snip>
    </snip>
  </frame>
</frameCustomisation>
```

Frame to be customised

Specification to be generated and validation schema

Component to be generated and validation schema

Customisations to be applied to frame 'DummyFrame.xml'

Next frame to be customised (elided)

Figure 3-12 – Example Frame Customisation

We use the Xalan XSLT processor to apply the generic transformation rules of the script `GenerateSpecification` to the specified frame customisation file (see Figure 3-13). Since we require access to a number of files (the customisation file and the frame(s) to be customised; these are referenced by the customisation file) it is necessary to use Xalan's `Redirect` extension; however, this is the only non-standard XSLT feature we have currently found necessary.

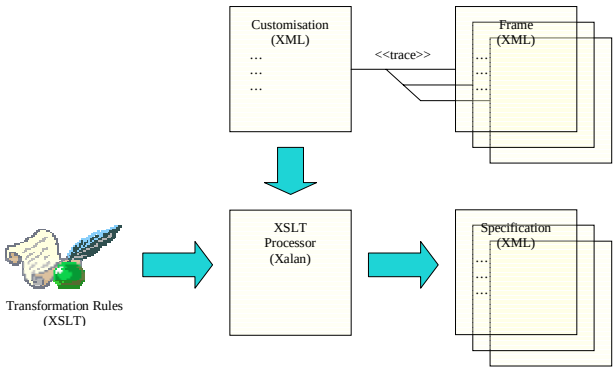

Figure 3-13 – Generation of Specifications from Frames

The specification generated as a result of this process is really only an intermediate stage; it comprises a copy of the frame being customised merged with the customisations to be applied (see Figure 3-14).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE specification SYSTEM "HumptyDumptySpecification.dtd">
<specification xmlns:dbg="http://www.baesystems.com/XSLT/namespace/debug">
    <document name="Frame">
        <preamble>
            <file name="DummyFrame.xml"/>
            <description>This is a test file.</description>
        </preamble>
    </document>
    <document name="Customisation">
        <preamble>
            <file name="DummyFrameCustomisation.xml"/>
            <description>This is a customisation test file.</description>
        </preamble>
    </document>
    <frameStyle type="product"/>
    <component name="/home/chris/Frame_Technology/Test/HumptyDumpty.xml"
               dtd="/home/chris/Frame_Technology/Test/HumptyDumpty.dtd"/>
    <customisations>
        <customisation tagName="lead"
                       attributeName="name"
                       value="HumptyDumpty"/>
        <customisation tagName="case"
                       attributeName="ending"
                       value="gruesome"/>
        <customisation tagName="case"
                       attributeName="epitaph"
                       value="burialAtSea"/>
    </customisations>
    <body>
        <contents>
            <line>
                <lead name="HumptyDumpty"/> sat on a wall
            </line>
            <line>
                <lead name="HumptyDumpty"/> had a great fall
            </line>
            <line>
                All the <monarch name="King"/>'s horses
                and all the <monarch name="King"/>'s men
            </line>
            <line>
                Couldn't put <lead name="HumptyDumpty"/> together again.
            </line>            <case name="ending">
                <when test="happyEverAfter">
                    <line>
                        But it's OK, <lead name="HumptyDumpty"/> recovered.
                    </line>
                </when>
                <when test="irony">
                    <line>
                        Unfortunately, <lead name="HumptyDumpty"/> cracked-up.
                    </line>
                </when>
                <otherwise>
                    <line>
                        Poor <lead name="HumptyDumpty"/> died.
                    </line>
                </otherwise>
            </case>
            <case name="epitaph">
                <when test="burialAtSea">
                    <line>
                        <lead name="HumptyDumpty"/> was commended to the sea!
                    </line>
                </when>
                <when test="burial">
                    <line>
                        <lead name="HumptyDumpty"/> is now six feet under!
                    </line>
                </when>
                <otherwise>
                    <line>
                        Sadly, <lead name="HumptyDumpty"/> was cremated!
                    </line>
                </otherwise>
            </case>
        </contents>
    </body>
</specification>
```

Figure 3-14 – Frame Specification (example)

Although the intention of this approach was as an aid to debugging, it may ultimately prove to be beneficial as a form of intermediate frame storage whereby the extent of the customisation could be extended by subsequent customisation passes. However, if we were to investigate such a multi-pass framing strategy, we would be inclined to apply the frame document model such that frames beget frames. Such an approach may help to provide a platform on which to support efficient use of the late code injection feature described by Völter in [22], and support for a more feature-oriented customisation strategy (where features tend to crosscut components).

Having generated the specification, we then perform a relatively simple transformation to actually apply the customisations to the frame to generate the product. In the case of the example in Figure 3-14 this entails the substitution of the name attribute of the element `<lead>` and evaluation of the two case statements. We currently support only the simplest of tests, however one could support arbitrarily complex expressions at the expense of added complexity in the transformation script (e.g. parsing and evaluation). Figure 3-15 provides an example of the product generated from the specification shown in Figure 3-14; from this it can be seen that the transformation into an HTML or text document should be quite simple, although we still need to propagate the frame style element from the specification. If the product document were to be another customisation document, then it would be necessary to apply the generic customisation schema to the generated XML document. We did not investigate this transformation, although we know the transformation of documents to conform to a differing schema to be feasible [20], our aim would be the unification of the specification and frame artefact.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE product SYSTEM "/home/chris/Frame_Technology/Test/HumptyDumpty.dtd">
<product xmlns:dbg="http://www.baesystems.com/XSLT/namespace/debug">
    <line>
        <lead name="HumptyDumpty"/> sat on a wall
    </line>
    <line>
        <lead name="HumptyDumpty"/> had a great fall
    </line>
    <line>
        All the <monarch name="King"/>’s horses
        and all the <monarch name="King"/>’s men
    </line>
    <line>
        Couldn’t put <lead name="HumptyDumpty"/> together again.
    </line>
    <line>
        Poor <lead name="HumptyDumpty"/> died.
    </line>
    <line>
        <lead name="HumptyDumpty"/> was commended to the sea!
    </line>
</product>
```

Figure 3-15 – Simple Product (example)

4 Applying Frame Technology

In [23] McCarthy defines Mass Customisation (MC) as:

> *‘The ability to provide customised products and services to individual customers or niche market segments on a large scale, without losing the benefits of mass production.’*

He asserts that MC should be considered in circumstances where the customer values the ability to discriminate and differentiate between products based on selectable features. Frame Technology provides an environment that may support such aspirations, whilst being naturally sympathetic to the library scalability problem as described by Biggerstaff [11]; since a single frame may support the generation of a plethora of concrete components. Similarly, the ability to adapt frames by injecting code into predefined locations also allows for customisation beyond the bounds that may have been envisaged at the time of creation. Language independence makes the technology amenable to the framing of any text-based information, not just executable code, although, naturally, this is one of the more appealing applications. Frame Technology may also be an attractive alternative to the complete reengineering of legacy applications into the latest design notation, with subsequent forward engineering. In such circumstances where time and/or funding is short, or where there isn’t the longevity in the project to warrant reengineering, the wrapping of legacy products within frames may be a pragmatic alternative, to facilitate the adoption of a PLA. Another possible motivation for adopting frame technology could be in areas where strict performance constraints are to be applied (e.g. run-time or memory footprint, or both). In such circumstances code generation from a model may be inefficient, or require significant hand optimisation, hence, the assembly of optimised components from frames *may* prove to be more effective. However, we note that the focus of a project tends to evolve over time, usually starting with a drive to maximise functionality to gain market penetration and customer confidence, then moving, as the product matures, towards the optimising of packaging and resource requirements. Frame Technology would also appear to lend itself well to the integration of cross-cutting concerns [22], such as the adoption of some particular style of Built-In Test philosophy, which might require the adaptation of many components. However, we wouldn’t wish to give the view that frame technology is a panacea, rather it is one of a number of possible approaches one might adopt, depending on one’s starting point, and one’s goals.

In [22] Völter contrasts a number of code generation technologies and summarises frame technology as being difficult to learn, but both flexible and suitable for complex uses. As with any family-based approach, the ultimate success will also depend largely on getting the domain analysis right and choosing to model the right features at the right level of abstraction. One possible reason for the apparently steep learning curve is that frame technology is really a form of imperative programming, the framed component must be decorated with frame commands, and the scope of the effect of customisations can be difficult to identify. This situation would be exacerbated if frames were to be adapted multiple times. A higher level view is required to enable the frame developer and/or user to visualise the component relationships and interdependencies; typically, the PLA provides this role. As an example, we could provide a succinct visual model of the family of components expressed by the frame described in Figure 3-8 and Figure 3-10 in the form of a feature diagram following the metamodel described in [2], see Figure 4-1.

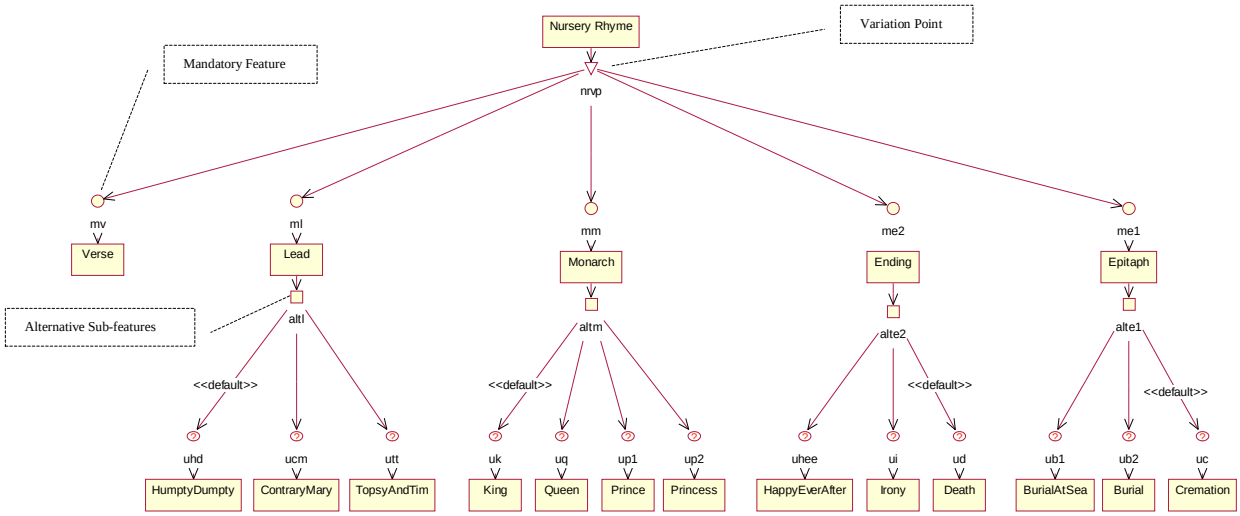

Figure 4-1 – Graphical Representation of Frame

Whilst we have elided some constraints, it may be seen from the feature diagram that, if no customisations are made, a nursery rhyme will be generated featuring Humpty Dumpty as the lead, King as the monarch, Death as the ending, and Cremation as the epitaph. It can also be seen that such a simple frame actually embodies 108 possible variations of nursery rhyme[11], although, clearly, not all possible variants may be legitimate; hence additional constraints would be placed on the feature model.

The writing of frames will also be more demanding than the writing of conventional software, since a frame contains a variety of products. Whilst we have investigated only simple frames, we believe that the use of tools would be imperative in all but the most trivial of cases, such tools may take the form of context sensitive editors, frame generators (wrappers), etc.

5 Related Work

The domain of code generation is a very vibrant and diverse area of work in which the potential rewards are great. The typical approaches to code generation may be considered as falling into one of two general categories:

1. Assembly of a system from pre-existing fragments – the compose and adapt approach
2. Generation from a higher level abstraction (e.g. a UML model) – the transformational approach

Clearly, Frame Technology falls into the former category. In this section we will try to contrast our approach to Frame Technology against a number of popular approaches to code generation.

5.1 Generative Programming

Generative Programming is a fairly recent contribution to the code generation toolbox, and has been popularised by (amongst others) Czarnecki & Eisenecker; it is defined as [24]:

> *'A software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customised and optimised intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.'*

[11] 3 leading characters * 4 monarchs * 3 endings * 3 epitaphs = 108 variations

Generative Programming relies on template metaprogramming using a suitable C++ compiler; the template metaprograms contain embedded generators which are executed by the C++ pre-processor before the compilation stage commences to adapt the code that is ultimately compiled and linked. Typically, GP techniques may be used to select specific classes and/or optimisations from configuration knowledge, resulting in the generation of a customised component. An example of static template metaprogramming is illustrated in Figure 5-1 by way of a simple factorial program (taken from [24]).

```
// Static metaprogramming example
#include <iostream>

using namespace std;
//
// compile time factorial function
// n>0
template<int n>
struct Factorial {
  enum { RET = Factorial<n-1>::RET * n };
};

// n=0 (terminating case)
template<>
struct Factorial<0> {
  enum { RET = 1 };
};

int main()
{
  cout << "Executing" << endl;
  cout << "Factorial<7>::RET=";
  cout << Factorial<7>::RET << endl;
  return 0;
}

exection of function …
$ factorial
Executing
Factorial<7>::RET=5040
```

Figure 5-1 – Compile Time Factorial

In this example, the result of the factorial expression is computed at compile time, in contrast to the normal approach where a recursive factorial function is evaluated at run-time (or during elaboration). This is an example of using C++ as a two level language, comprising both static code (metaprogram) and dynamic code (regular C++) [24].

The aims of GP are similar to those of Frame Technology, to address the library scalability problem by supporting the generation of customised components on demand from a set of reusable components. However, GP is aimed specifically at the C++ market and, whilst it does not require any non-standard language features, may make extreme demands on the compiler, leading to variable results across available compilers (see, for example, [25]). It may also be seen from Figure 5-1 that the style of the template metaprograms is somewhat different to code executed in the run-time domain, and is more functional in style. However, in a relatively short space of time, GP has acquired a significant following and been used in a number of domains to construct additional C++ libraries, such as:

- Compile time dimensionality checking in SI Units [25]
- High performance tensor library [26]
- Matrix template library [24]
- Blitz++ scientific computing library[12]

On the down side to GP, we have already noted the lack of standardisation of the C++ compilers, however, Czarnecki & Eisenecker also highlight the lack of adequate debugging support and error reporting for development of template libraries; in fact, they refer to this as a severe limitation [24]. This limitation may also be levelled at the frame processor, although we do have access to its internals and are not bound to a compiler vendor; instrumentation of the XSLT-based frame processor is also quite simple (see [20]).

5.2 Template Language

In [27] Cleaveland describes the use of Java Server Pages (JSP) and XSLT in the design of program generators, concluding that each approach has its own drawbacks as neither was designed with the explicit aim of supporting program generation. It is also worthy of note that, in his demonstration of code generation via XSLT, the author embeds code fragments within the XSLT scripts, and, hence, would require scripts for each application domain and programming language, although the specification would (conceivably) be reusable. Whilst we would agree with some of the shortcomings of XML/XSLT identified by Cleaveland (e.g. both are somewhat verbose), we do not find the Template Language (TL) to be an entirely adequate solution. This is in contrast to our approach of providing a set of generic transformations and describing the program family with the XML specification. The author describes a solution to the problems he perceives with the use of JSP and XSLT in the form of the TL. The TL is designed against 11 goals, although some of the goals are more syntactic in nature and appear to relate to pretty-printing of the generated code, and is based on a Java implementation with abstract program specifications supplied in XML documents. Transformations

[12] http://www.oonumerics.org/blitz/

are performed on the XML specifications via the DOM (i.e. XML is transformed via a Java interface to an XML reader, rather than via XSLT). It is also noted that the author provides a custom API to provide access to the XPath language in order to allow the identification and selection of specific nodes[13], and an escape mechanism to allow the execution of Java statements from within the TL via interpreter or compiled approaches. Ultimately, it appears that the TL simply reduces the semantic gap between the code generator specification and the target language. We illustrate the major difference in approach between the TL and Frame Technology in Figure 5-2.

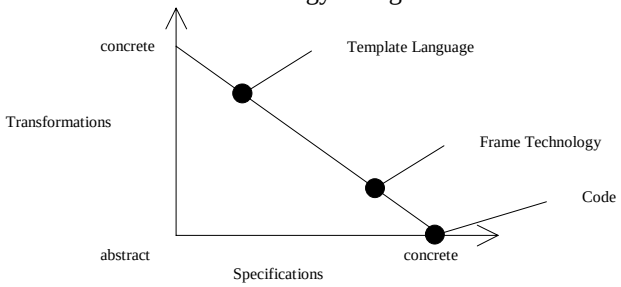

Figure 5-2 – Template Language v. Frame Technology

5.3 Aspect Oriented Programming

Aspect Oriented Programming (AOP) is intended to provide strong support for the separation of concerns, such that crosscutting features (aspects) are themselves modular units, rather than pervading many classes as is often the case with conventional imperative and object oriented languages. In addition to *aspects* (non-functional components), AOP offers the traditional structuring units of programming: hierarchical modules, objects, and procedures (functional components). Aspects could be viewed as providing a third dimension to the static view of software architecture (see Figure 5-3); although this third dimension may be identified as a result of the collaborations that are evident from the dynamic view of the system. In AOP, collaboration patterns are referred to as *emergent entities* and may include data flow and concurrency patterns [28].

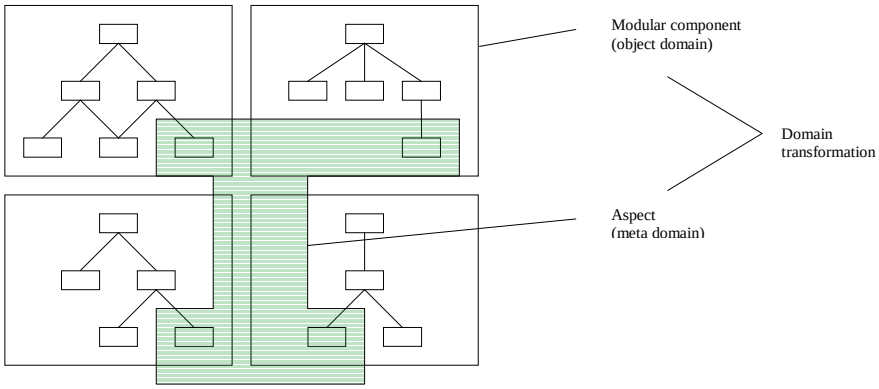

Figure 5-3 – Aspect Oriented View of Software Architecture

The aspect-oriented approach offers advantages to software reuse as functionality will tend to be separated and packaged more cleanly. Maintenance is similarly eased as the design and implementation remain a faithful reflection of the problem domain, and the need for refactoring is reduced (e.g. in response to performance requirements). Hence, AOP should help to resist the architectural rot that is often evident in long-lived systems, where frequent changes render the software apparently ossified and brittle with age. The application is generated via an *aspect weaver* that fuses together *aspects* and *functional components* of the design/code expressed in an aspect language (e.g. AspectJ™) to produce a *woven* program. The woven program will include the cross-cutting aspect code, i.e. the separated concerns are now merged, however the design and implementation were developed from cleanly separated classes and entities. In [24], Czarnecki and Eisenecker provide an interesting view of a number of common design patterns cast in terms of aspects, such as the *mediator* pattern [31] which describes object interaction. The authors also describe a number of problems that can be encountered by the introduction of some patterns into conventional OO-based implementations (e.g. the *self* problem).

AOP has some resonance with Frame Technology; both approaches generate code by composing more primitive components. However, the view taken of these components is different; Frame Technology focuses on the *same-as-*

[13] Use of XPath is implicit with XSLT.

except view and is more open-ended, whereas AOP seeks to provide modularity in terms of classes and emergent entities. The implementation of cross-cutting concerns may be effected via Frame Technology, although this is really limited to the late code injection feature or a priori knowledge of such aspects, and is predicated on the availability of suitable break points in the frames to be adapted. Development of an aspect weaver requires a metamodel defining the syntax and semantics of the language to be woven; this is non-trivial task. It also requires the specification of the semantic relationship between the components [22].

5.4 Intensional Markup Language

The Intensional Markup Language (IML) is a language to extend HTML to provide support for the generation of multi-version web pages via parameterisations, the aim being to minimise the costs associated with the maintenance of multiple copies of similar web pages [29]. IML was developed from the author's previous experience of IHTML, which was not a programming language as such; lacking support for modularity, evaluation of expressions, and some language constructs. Parameterisation is used to control the generation of web pages in terms of data content and rendering style, and support the linking of shared web pages in diverse forms, rather than the *one size fits all* approach that is typical. The IML is implemented in the Intensional Sequential Evaluator (ISE); a Perl-like scripting language incorporating a run-time parametric versioning system. This allows all entities in the language to be versioned; hence programs executing within a *context* know implicitly which versions of the entities to use. The example ISE code in Figure 5-4 is taken from [29].

Parameters shown in bold:
lg = language
sx = gender
nm = name

```
#!/usr/bin/ise
print("Content-type: text/html\n\n");
print("<html> <title>Page</title> <body>\n");
$happy<><lg:en> = "happy";
$happy<lg:fr> = "content";
$happy<lg:fr+sx:fe> = "contente";
vswitch{
<lg:fr>          {
                  print(@[Bonjour, ##nm##, es-tu $$happy$$]@);
                  print(" avec cette page Web?\n");
                 }
<lg:en>          {
                  print(@[Hi, ##nm##, are you $$happy$$]@);
                  print("with this web page?\n");
};
print("</html>\n");
```

Figure 5-4 - Example ISE Program

Hence, this program will generate a simple web page customised by the value of language (`ln`), gender (`sx`) and name (`nm`) supplied to the web server (a simple plug-in is required). Existing web pages may be converted into ISE programs by the application of a document wrapper (the author demonstrates this two macros written in troff[14] to top and tail the source document). The author eschews the use of XML (and presumably XSLT) due to the lack of availability of tools at the time of writing (2000).

There are strong visual and logical similarities between the IML and Frame Technology; one can view the ISE programs as being analogous to frames, and the ISE as analogous to the Frame Command Language. The main differences between the two technologies would appear to be the absence of a late code injection feature in the ISE. Both technologies aim to generate components in response to a specification; in the case of Frame Technology, this is provided by the SPC, whereas, in the case of IME, this is provided via parameters to the web server. However, it is not clear how the page variability interface is advertised to the client, or even if it is. Whilst the stated intention of the IML is to support the generation of HTML pages from a family-oriented view, the IME would appear to be application independent, this isn't really surprising as the IML was itself developed from LEMUR a version control system for programs written in C [30]. In LEMUR, the authors define a refinement relation $\Re$ to define how one version is derived from others within a given context, and note that the set of all possible versions under $\Re$ forms a lattice, as we see in the frame/sub-frame structure.

5.5 Other Approaches to Frame Technology

In this section we present a summary of a number of recent approaches to the implementation of Frame Technology using the XML. Each approach stems from the foundation presented by Bassett in [9], and feature in common the general theme of expressing frames in an XML meta-model, and operating on the frames via a bespoke frame processor.

5.5.1 XML Variant Configuration Language

Wong, et al. identify the tight coupling between the Fusion™ toolset and the COBOL programming language as a motivation for the recasting of frame technology in a more open and domain-independent approach [13]. Furthermore,

[14] A unix formatting and phototypesetting program dating from the 1970's.

the authors assert that frame commands, as described by Bassett, do not address the needs of OO and CBD-based design approaches. The aim of the XML Variant Configuration Language (XVCL) is to support investigation of the application of frame technology within different environments, primarily OO-based projects. The authors describe the use of XVCL within the context of the development of a family of Computer Aided Dispatch systems[15]. The authors recast the frame command language in terms of XML elements, and instrument artefacts for adaptation in XML documents, termed *x-frames*, using the frame commands. Frame Processing is performed by an XML-aware frame processor; this is written on the back of JAXP, a Java implementation of the SAX interface[16]. The scope of customisations covers all subordinate frames, as described in [9], the stated benefit being the context-independence of intermediate frames. The notion of logical collections of frames in the form of a modular component (*x-package*) is described. The relationship between *x-packages* and *x-frames* is many-to-many, and the authors illustrate their use of this feature to provide stakeholder views of the system; this relationship resonates with *aspects* in AOP, although the *weaving* is limited to the `import` and `copy` directives. The authors report that this feature supports management of the namespace, but do not identify why use of the XML namespace feature is deprecated.

Whilst the authors catalogue their mapping from Bassett's frame commands to XVCL, they do not describe any form of content model to support validation of the *x-frames*. XVCL extends Bassett's frame commands, most notably with the introduction of *x-packages*, language-specific support is also introduced via XML attributes; this is used to support the pretty printing and destination of output files. Furthermore, whilst one of the stated aims of the exercise was to investigate the application of Frame Technology to modern development paradigms, the authors provide little feedback on how frame technology affected their design and vice versa, concluding that further exploration is required. Finally, it is curious that the authors choose to use the Fusion toolset in preference to XVCL to support the Flexible Variant Configuration (FVC) tool [32].

5.5.2 Frame Oriented Programming

Sauer describes Frame Oriented Programming (FOP) as supporting both the modularising techniques of OOP and AOP to any text-based artefact through the application of template driven code generation [21]. The author provides an outline of a Frame Processing Language (FPL) that is based heavily on that described by Bassett [9] following the syntax provided by Jarzabek et al. for use in an XML environment (e.g. see [13]). The FPL, as described, differs little from Bassett's original description, and the focus is more on the representation of the FPL in the XML, although there is no evidence of the grammar of the language being enforced via the application of the DTD or XML Schema. The author provides a simple example of the late code injection features supported by the FPL and describes the precedence rules applied. The scope of this feature is described as the set of all subordinate frames, i.e. if frame `A` adapts frame `B` and frame `B` adapts frame `C` then frame `A` may also adapt parameters in frame `C` directly, again, this is in accordance with Bassett's frames [9]. Similar scoping rules are used to support the customisation of variables, a combination of frame adaptation and variable assignment commands will support both broadcasting and narrowcasting models as described in [9]. The FPL supports escapes to the Java programming language to allow the integration of custom functions (written in Java) into the FPL, as an example the author provides a function to convert between package (dotted) notation and directory notation. It is interesting to note that Cleaveland identifies the programming language escape mechanism in a list of 11 design goals for the Template Language [27].

Sauer describes the use of FPL within the context of tool set supporting the generation of Java code and XML schemas from metadata descriptions (XMI). The XMI metadata is transformed into FPL via an XSLT stylesheet, from this point on a number of common frames are used to generate the code based on a prototypical structures. The common frames described provide templates to enforce and allow extension of (e.g.):

- Class structure
- Attribute structure
- Operation structure
- Class associations

This approach is very similar to that taken by the tool Scriptor[17] produced by Sodifrance. Scriptor reads in (e.g.) a UML model via XMI® and processes the model elements (it has implicit knowledge of the underlying metamodel) according to user-defined templates/transformations somewhat akin to frame processing at the meta-level. This style of code generation is classified by Völter as the *Templates + Metamodel* pattern [22].

[15] CAD systems are intended to support the dispatching of emergency services to incidents.
[16] SAX is an event-oriented interface supporting the provision of bespoke event handlers.
[17] A code generator generator.

Sauer's approach to Frame Technology appears to be aimed at the application of Model Driven Architecture (MDA) technology by way of separating the UML model from the underlying implementation infrastructure, such as messaging protocols. Hence, the UML model is used to generate the Platform Independent Model (PIM), whilst the Platform Specific Model (PSM) is generated by the frame processor injecting the PSM-related data from *aspectual* frames, illustrating synergy between FOP and AOP.

The frame processor is written in CLOS[18] and performs a one-pass depth-then-breadth process stemming from the SPC (this is in keeping with Bassett). Sauer reports some restrictions relating to the implementation of the FPL frame processor, most notably the absence of support for infix expressions, and the requirement to break out of CLOS environment into Java to perform certain functions; functions that would be easily achievable in XSLT. It is unclear whether the use of Java break-outs is a pragmatic approach or an indication of a lack of flexibility of the frame processor and FPL, furthermore the use of the XML namespaces is identified, however its integration is unclear. Contrasting Sauer's approach against that of Bassett [9] one can see many strong similarities, almost to the extent that it is tempting to classify FPL as simply Frame Technology re-cast in XML with a few minor additions and refinements. However, this would be to loose sight of Sauer's goals; Sauer demonstrates an integrated approach to the support of a PLA (applications for the insurance industry) using commonly available technologies within which frame technology plays the central coordinating role.

5.5.3 The PoLITe Project

The PoLITe project is being undertaken by the Faunhofer IESE, and is investigating product line technologies from three perspectives:

- Configuration management
- Component technologies
- Generative features of programming languages

In [33] Patzke & Muthig report on a case study to investigate the implementation of a PLA supported by Frame Technology. The authors report on the differences in a frame-based approach versus the conventional OO-based approach; they also summarise a number of frame-related idioms identified as the project progressed. The case for a product line approach is couched in terms of reduced development and maintenance effort, and Frame Technology is chosen to investigate the management of component commonality and variability required to support the fabrication of customised artefacts; rather than the *one size fits all* approach. The authors concur with Czarnecki & Eisenecker [24] that design patterns such as the Adapter and Decorator patterns [31] may be applied to support the reuse of components *as-is*, but at the cost of the possible introduction of undesirable complexities. Hence, the reuse paradigm is extended to cover adaptation (*same-as-except*) and reuse (*as-is*).

In order to assess the use of Frame Technology, the authors use a simple frame processor (*fp*) supporting only a few frame commands; hence it is far less capable than XVCL, FPL, or Fusion™. An example program family is expressed in Python and framed in the (restricted) *fp* command language to demonstrate the effect of frame processing, and the simplification offered over the management of component customisations introduced into a conventional OO design. The difficulty of removing behaviour in a conventional OO design is illustrated via the (generally discouraged) practice of introducing null bodies into concrete operations in variant subclasses. The effect of cross-cutting requirements on conventional OO designs is also demonstrated by the introduction of relatively large amounts of similar code across many disparate classes. The effect of the various changes on both the conventional OO and frame-based approaches is summarised, demonstrating that Frame Technology results in both reduced code growth and significantly smaller implementations. This serves to highlight the robustness of the frame-based approach in the face of evolution of the product line, contrasted against the need to refactor the conventional OO design in response to seemingly minor variations in requirement. Finally, the authors provide an estimate that the development effort required using Frame Technology yielded a saving of 13% when compared to the conventional OO approach, and conclude that Frame Technology offers significant advantages over OO for implementing product lines.

5.6 Auto Code Generation

Automatic Code Generation (ACG) from higher level abstractions (e.g. models) is an appealing approach to providing significant improvements to productivity; however, this requires the bridging of a semantic gap. A UML class diagram such as that shown in Figure 5-5 provides a view of a number of constituent classes of an Earth Model component, however this static view is clearly some way from (e.g.) Ada code. In order to bridge the semantic gap from the static and dynamic views of the UML model to Ada code we must perform a number of transformations according to a known set of mapping rules (e.g. a UML class is generally mapped to an Ada type contained with a package). Such mappings

[18] Common LISP Object System – an OO extension to Common LISP.

(profiles) will be unique to a specific language, and may vary to suite particular flavours of languages as applied to particular domains (e.g. run-time dispatching may be discouraged in safety critical applications). Furthermore, some application domains (e.g. avionics) may require rigorous arguments to be made about the integrity of the code produced by the generator [34], placing yet more requirements for precision in the transformation/mapping rules used to get from the model to the implementation. Such requirements run counter to the typical approach by tool vendors of treating the code generator as a black box.

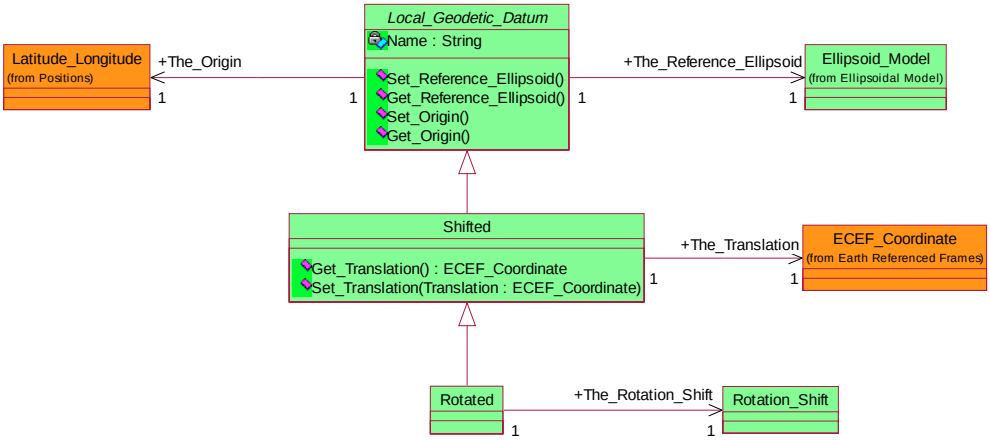

Figure 5-5 – Earth Model (example)

There are clearly some significant differences between Frame Technology and ACG; most notably, the *translation* from the abstract to the concrete in contrast to the *assembly* of a product from prefabricated parts. ACG may also be viewed as a forward engineering process, moving the reuse issue from the implementation to the design level, thereby rendering the reuse issue language independent; at the cost of providing semantically equivalent mappings to support design to code transformations. However, the metamodel-based transformational approach of applying code templates advocated by Audsley et al. in [34] bears some striking similarities to the frame based approach of Sauer in [21], although the ultimate aims are subtly different; i.e. trusted code generation compared with separation of the PIM and PSM.

5.7 Classification of Approaches to Code Generation

We have discussed a number of popular approaches to code generation. Table 1 provides a summary of these approaches and identifies the strengths and weaknesses perceived.

Approach	Category	Strength	Weakness
Generative Programming	Compose & Adapt	• Generation of highly optimised components • No language extensions required • No additional tools required • Domain independence	• Language-specific (C++) • Compiler maturity issues • Narrow domain applicability? • Two-tiered language (functional/OO) • Debugging support
Template Language	Transformational	• Generic specifications • Commonly available tools (XML/Java) • Domain independence	• Language-specific transformations • Lacks homogeneity – break-outs into underlying language (e.g.) Java • No evidence of commercial use • Debugging support
Aspect-Oriented Programming	Compose & Adapt	• Clean separation of concerns • No additional tools required	• Language specific (e.g. Aspect-J) • High learning complexity
Intensional Markup Language	Compose & Adapt	• Simplicity of ISE wrapping language	• Introduction of secondary language (ISE) • Domain specific (web page generation) • Debugging support
Auto Code Generation	Transformational	• Clear and unambiguous semantics required • Support for *trusted code generation* • Potential return on investment • Portablity (target architecture/MDA) • Ease of use (visual paradigm) • Tailorability	• Blue-sky approach • Required investment • Optimisation of code generated • Maturity
Frame Technology	Compose & Adapt	• Language independent • Leverage legacy assets • Investment required • Support for MDA approach • Domain independence	• Weak semantics of Frames • FCL wrapping language not well specified • Scope of effects unclear • Debugging support • High learning complexity

Table 1 – Code Generation Techniques

Völter identifies a number of stages in the software development life-cycle where program generation technologies may be applied; these technologies are cast in the context of when in the development process decisions may be made about certain features [22]. Although Volter's aim is the characterisation of a number of recurring patterns that may underpin

program generation, we can use Figure 5-6 (derived from [22]) to identify the applicability of the technologies we have reviewed.

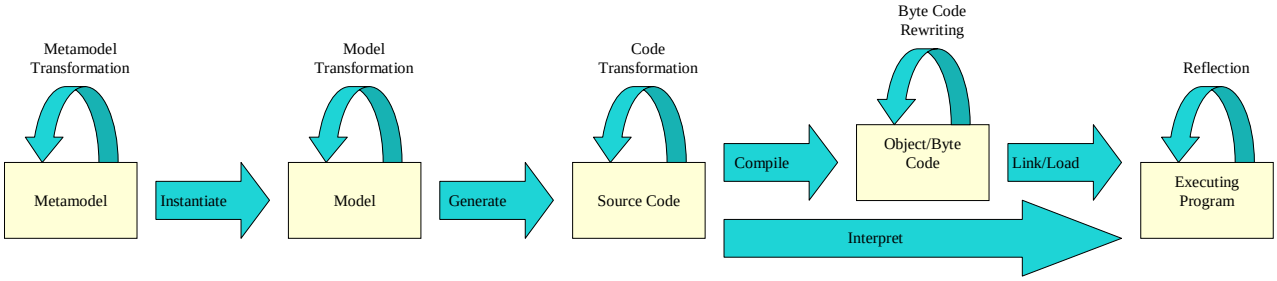

Figure 5-6 – Program Generation Opportunities

Using Völter's diagram to classify the code generation techniques discussed by applicability to phases of the life-cycle (see Table 2), we can begin to see some distinctions emerging. Each approach attempts to address the code generation problem at one (or more) specific points; a lack of support in certain areas is also apparent. The decision of which technology to use may be based on many factors, such as the binding time of certain features [22], and also the representation of the underlying data model, legacy issues, product certification issues, etc.

Phase	FT	ACG	TL	GP	IML	AOP
Metamodel Transformation						
Model Instantiation						
Model Transformation	✓					
Source Code Generation	✓	✓	✓			
Code Transformation				✓	✓	✓
Byte Code Rewriting						✓
Reflection						✓

Table 2 – Applicability of Techniques

6 Meta Frames

One of the weaknesses inherent in the approaches to the implementation of Frame Technology described above is considered to be the apparent disconnect between the Frames and the model of the components to be framed and hence the systems to be generated. Whilst this may be partly due to the wrapper-like nature of frames, which might be considered to be useful if we're trying to frame legacy components, it may be argued that it is not in the spirit of the MDA approach of *modelling everything*. It can be seen from the summary of the work on frames, that whilst a schema may be defined (e.g. in the form of a DTD or XML schema), there is generally no visual model, and we tend to be restricted to the concrete syntax[19]. Furthermore, it may be argued that the current approaches to Frame Technology are generally platform-specific, even if that platform is the ubiquitous XML, and, again, one sees a divergence between the specification of Frames and the framed components. Hence, we suggest that Frame Technology could be made more accessible if it were to be lifted to the MDA space and specified in the same language as the components it may be used to frame. In order to achieve this within the context of the UML (and MDA) it is necessary to develop a metamodel of Frames such that this metamodel could be embedded within a modelling environment. Following on from our earlier investigation of Frames in an XML/XSLT environment, we are in the process of developing a metamodel of Frames – *meta-Frames*. We have expressed our metamodel using the modelling tool XMT [6], this tool allows the metamodel to be checked against constraints (invariants) and to be exercised (executed) in the form of snapshots (object diagrams describing specific instances); it is also possible to exercise the model via a command line interface. Our initial model of *meta-frames* is shown in Figure 6-1 and currently supports the use of Break commands and various forms of Copy command, including extension and management of the context of execution of the Frame.

[19] Although we note that some commercial XML tools may provide a graphical view of the schema, e.g. DTDChart from Intelligent Systems Research, see www.instsysr.com/dtdchart.htm

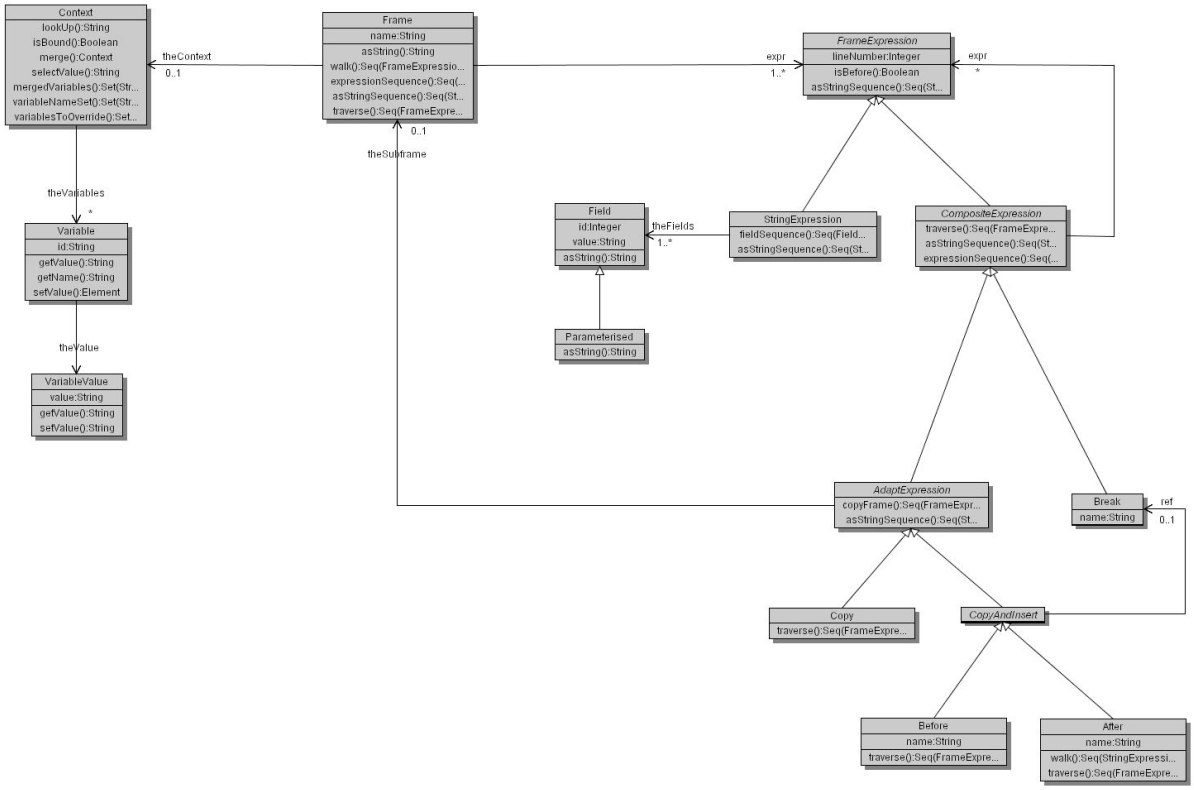

Figure 6-1 – A Metamodel of Frames

Borrowing from one of Bassett's examples [9] we cast a simple frame in terms of the snapshot in Figure 6-2.

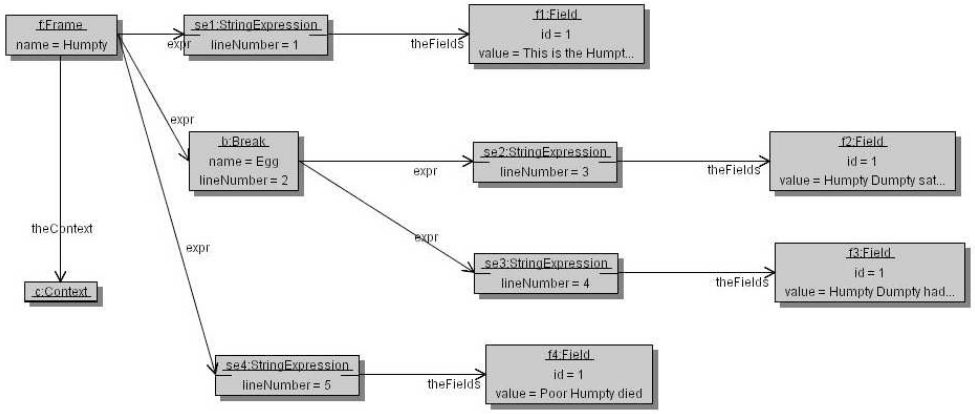

Figure 6-2 – Example Frame with Break Expression

Processing of this frame (via the operation `asString()`)yields the output in Figure 6-3.

```
XMT> FrameWithBreak::f.asString();
This is the Humpty Dumpty archetype
Humpty Dumpty sat on the wall
Humpty Dumpty had a great fall
Poor Humpty died
```

Figure 6-3 – Result of Processing Frame 'Humpty'

Finally, in Figure 6-4 we provide a slightly more complex example in which one frame (f_2) uses and adapts another frame (f_1). In this example it is necessary for the client (f_2) to make a copy of the subordinate frame (f_1) and to execute the copy within a local context.

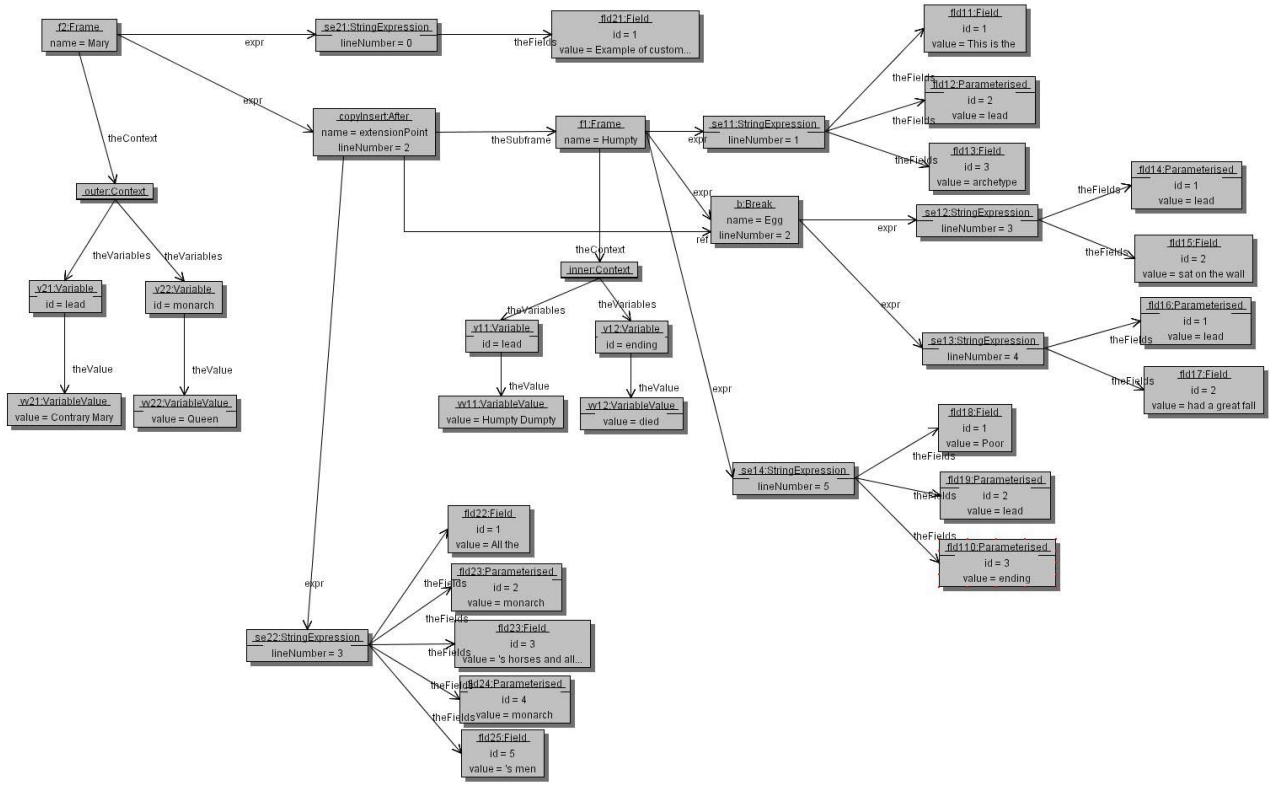

Figure 6-4 – Example Frame Customisation

In the above example frame f_2 executes within the context c_2, whereas frame f_1 has the default context c_1. The semantics of the Copy expression are such that the copy of frame f_1 must be executed within the context $c_{new}=c_2 \oplus c_1$, i.e. the context c_2 takes precedence over the context c_1. Hence, the Copy expression does not apply changes to the copied frame, only the context in which it is executed. Processing of frame f_1 within the context c_1 (innerContext) yields the output in Figure 6-5:

```
XMT> FrameCopyAndAdapt::f1.asString();
This is the Humpty Dumpty archetype
Humpty Dumpty sat on the wall
Humpty Dumpty had a great fall
Poor Humpty Dumpty died
```

Figure 6-5 – Results of Processing Frame f_1 ('Humpty')

Whereas processing of the frame f_2 within the context c_2 (outerContext) causes frame f_1 to be processed within the context $c_{new}=c_2 \oplus c_1$, and yields the output in Figure 6-6:

```
XMT> FrameCopyAndAdapt::f2.asString();
Example of customise and extend frame adaptation
This is the Contrary Mary archetype
Contrary Mary sat on the wall
Contrary Mary had a great fall
All the Queen's horses and all the Queen's men
Poor Contrary Mary died
```

Figure 6-6 – Results of Processing Frame f_2 ('Mary')

We have given a brief description of our metamodel for Frames. Whilst this is currently text-based, in accordance with all other reported implementations of Frame Technology we have found, and provides us with some confidence of its correctness, we do not believe that framed elements should necessarily be restricted to text. We intend to investigate the extension of our metamodel to support the framing of metamodel elements such that Frame Technology could be used to support the custom development of modelling languages from a kit of parts. It is our intention to try to use

meta-frames to provide metamodel support for the precise modelling of features and PLAs in the UML. We envisage a recursive relationship between *meta-frames*, PLAs and the metamodel such that custom languages could be developed with relative ease from a defined kit of parts; this idea is illustrated in Figure 6-7 and is the subject of ongoing work to be reported at a later date.

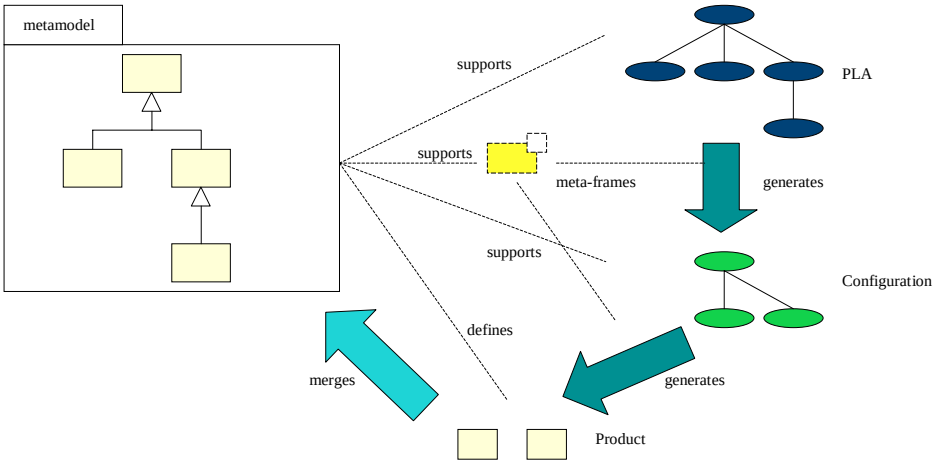

Figure 6-7 – Custom Language Generation

It should be noted that the components used (features) to build the PLA are underpinned with well-formedness rules to ensure the construction of valid PLAs. The PLA also contains configuration rules to ensure that only valid Configuration models may be generated, i.e. selection of compatible features. The *meta-frames* may be used to support both the generation of the Configuration model from the PLA and the generation of the target products from the Configuration model.

7 Conclusions

In 1968 McIlroy identified the need for *industrialism* in the software engineering domain [35]. Recognising the duality of the demand to meet the needs of individual customers and the need to maintain the cost and productivity benefits of mass-production, he called for a software components sub-industry. This goal has been achieved to varying degrees in some domains, e.g. operating systems and math libraries, however the value chain specialisation[20] we see here is not typical of the more context-sensitive domains. The *same-as-except* paradigm that underpins Frame Technology is novel and extremely powerful. Judicious use of frames may support the application of OO design principles to non-OO implementation languages, or more restrictive language profiles. Furthermore, the ability to suppress redundant features and inject new features provides robustness to the PLA, resisting the need to refactor unnecessarily, and leads to significantly more compact products [33]. Frame Technology is reported as providing productivity increases over conventional methods, including OO [9] and [33], however it is also reported to incur a steep learning curve [33]. Frames may also be difficult to read and the scope of variables difficult to identify without the support of an intelligent editor. Although Frame Technology supports the evolution of product lines and adaptation of product line components; care is needed to define (externally) the composition rules implicit in the Feature Model. Hence, frame technology requires underpinning with a formal description of the component composition rules to validate configurations. For example, cross-links of the following form would not normally be embedded within the frame hierarchy, such constraints belong in the feature model.

- *feature X is incompatible with feature Y*
- *if feature A is selected then feature B must also be selected*

Hence, we view feature modelling and frame technology as complimentary technologies supporting an integrated PLA approach.

We have summarised a number of approaches the implementation of Frame Technology described in the literature and contrasted Frame Technology against a number of comparable technologies aimed at supporting the generation of custom artefacts from specifications. We have presented our approach to the implementation of Frame Technology using the freely available and ubiquitous technologies of XML and XSLT[21], and provided a contrasting view based on an equally open meta-modelling approach. Whilst difficulties were encountered with the implementation of an XML/XSLT-based frame processor, we found this to be a rewarding activity, providing us with a better insight into

[20] This is sometimes referred to as horizontal integration.
[21] An earlier implementation used XML and Python; using the SAX and, subsequently, DOM interfaces.

Frames. Ultimately, we believe our future work with Frame Technology will be at the platform-independent metamodel level, using *meta-frames* to support our wider aspirations of defining a meta-model to support PLAs in the UML.

We suggest that all approaches to the generation of custom artefacts fit the generic model provided by Cleaveland in [27] to a greater or lesser extent (see Figure 7-1). With regard to the template or frame-based approaches, the major differences appear to relate to the scope of the roles played by the specification and transformation rules (see Figure 5-2).

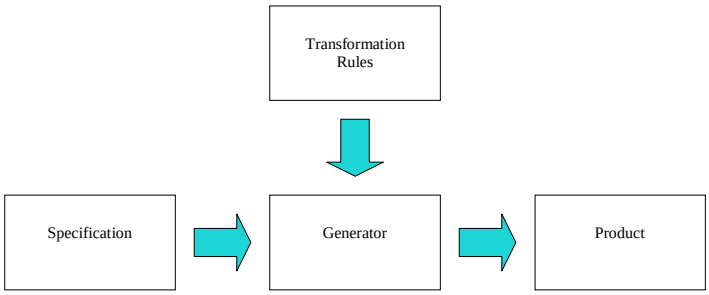

Figure 7-1 – Generic Code Generator Model

For example, in the design of the Template Language, Cleaveland trades compactness of XML specifications for language-specific transformations [27]. Frame Technology following Bassett's approach is kept language-independent by using a generic transformation language at the cost of a more verbose (and language-specific) specification (frame). Ultimately, the decision must be based on the market and tool chain anticipated, e.g. a family of XMI-based code generators could be supported by developing a number of language-specific transformations around a common generation engine operating on the XMI-based specifications.

Much work has been published on the application of the emerging discipline of *Generative Programming*, and a number of packages have been widely reported (see 5.1), in some cases these are freely available on the web. However, it can be seen that this work relates generally to relatively fine-grained components, e.g. math packages, albeit highly optimised and adaptable. Furthermore, in order to begin performing GP, all that is required is a good C++ compiler, although the programming style is somewhat different to that of the regular dynamic code. Aspect Oriented Programming has also received much coverage in the recent past; AOP is supported by (e.g.) an extension to the Java language in the form of Aspect-J. AOP continues to be an active area of research, however we see that the paradigm is bound closely to the implementation language.

7.1 Experiences with XML/XSLT

We were attracted to the use of XSLT in preference to a conventional programming language as we believed this would provide a more homogenous solution, e.g. XML and XSLT share a common syntax, and changes would be easy to prototype. We elected to use the XML (and the DTD) in preference to (say) *lex* for simplicity of use; our document models went through many revisions as our ideas and understanding evolved. This decision also incurs penalties in terms of precision, speed of execution, and verbosity. Use of the XML/DTD also appears better suited to the management of documents of a regular structure rather than arbitrarily nested documents as we defined; however, the pattern matching approach of XSLT is well suited to the task of transforming such documents. Casting our approach to the implementation of a frame processor in terms of the seven patterns for code generation identified by Völter [22], we are performing *model transformation* (a product family is transformed into a specific member) using the *Templates + Filtering* pattern. Whilst this approach is simple, Völter cautions that it may become very complex if the handling of more than trivial queries is required; in which case the *Templates + Metamodel* approach is advised. We feel that implementing support for controlling the scope of frame parameters in XSLT could be problematic. We are aware that such features are supported directly by some programming languages (e.g. common LISP – *special variable*[22]), and note the suggestion by some that XML/XSLT is *LISP by the backdoor*; although we would stop short of the analogy of XSLT to *LISP with angled brackets*.

We found the use of XML & XSLT useful as a prototyping language, and the use of the DTD provided some confidence that the generated artefacts were valid. However, the DTD is not as expressive as one might wish in a program generation environment; the syntax of the DTD is also very different to both XML and XSLT. As an example, consider the following simple Ada type declaration:

```
type Age_In_Years_Type is range 0 .. 150;
```

[22] http://c2.com/cgi-bin/wiki?SpecialVariable

Using the DTD syntax we would have to fully enumerate the type or introduce tags to represent the range constraints. Problems were also encountered when the DTD was combined with the use of the XML namespace feature. This situation is believed to be rectified by the introduction of the XML schema, however, at the time of writing there are no freely available tools beyond the beta-test stage. We found it necessary to use only one non-standard language feature in the transformation of our XML-based frames to control the navigation of multiple files, however this feature is provided by both the Xalan and Saxon XSLT processors. The use of XML is pragmatic as it is both human and machine readable, however it is somewhat verbose, and appeals for some form of language sensitive editor to allow the programming and editing of frames in a style more closely related to the domain language rather than XML nodes. This isn't seen as a major hurdle, and could possibly be accomplished via a plug-in to an extensible editor such as Emacs.

Whilst our XML/XSLT frame processor is very simple, implementing only a subset of the FCL as defined by Bassett in [9], it enabled us to gain a better understanding of Frame Technology, and there are clearly a number of improvements that could be applied. In hindsight, the separation of frames into frames and specifications was an unnecessary complication, introducing differing document models; although it proved useful as a debugging aid; this was partly a consequence of the peculiar semantics of `xsl:variable` which prevent the caching of document sub-trees prior to version 2.

7.2 Frames at the Meta-Level

Our intention is to use Frame Technology to provide variability at the metamodel level in support of our work in developing a UML metamodel of features and configurations [2]. In order to do this, we have developed a metamodel of frames (*meta-frames*) using the tool XMT; hence, we will have both a metamodel of frames and a metamodel of features, the semantic domain of which will be supported by frames. We also expect to use Frame Technology in its native role to realise a PLA case study (a family of navigation systems) as code. One of the themes that appears in all frame processor implementations is the problem of handling complex expressions; although some authors handle this by breaking out of the frame processor into the underlying programming language (e.g. Java). Instead of developing bespoke support for arbitrarily complex expressions in XMT (not a trivial task), we propose to use an extended form of the OCL. The OCL is well-defined [8] and is fully supported by XMT in the form of the eXecutable OCL (XOCL) and is compatible with the MDA view. The XOCL is essentially OCL with side-effects (an action language); this allows us to generate the frames directly from the tool (although we may need to develop a more amenable concrete syntax for the FCL). We view this approach as being faithful to our goal of openness, as it will be based on a precise specification (metamodel) of frames, we are also not aware of such work having been undertaken elsewhere; the norm being to bury the (implicit) metamodel in a 'closed' frame processor[23]. Jarzabek et al. report on their experience in handling variance in UML models with frame technology [32], however their approach to supporting the customisation of domain models requires the instrumenting of an XML representation of model components exported from a modelling tool via the XML Metadata Interchange (XMI®). We assume that some form of pre-processor is used to identify points of variability from predefined model element stereotypes, and instruments the XML description without human intervention. The XML file is then customised by the frame processor, and the resulting file read back into the tool; again, using the XMI (see Figure 7-2). The XMI is the standard tool interchange language, hence the approach is generally applicable; however there are problems with current standard of XMI, primarily the lack of support for graphical rendering of the model.

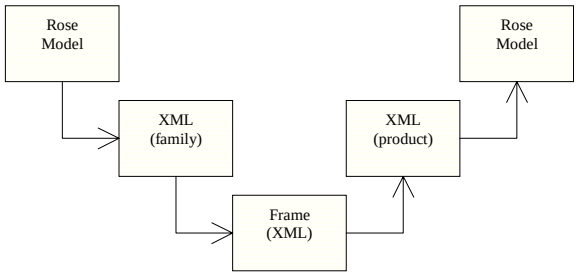

Figure 7-2 – Frame Processing of (UML) Rose Models

We believe that the FCL is not well specified, Wong et al. acknowledge the assistance of Netron Inc. during the development of XVCL for advice to ensure that frame concepts were correctly interpreted [13]. We believe this deficiency may be overcome by our robust and executable metamodel of frames.

[23] Although the frame processor may be freely available.

7.3 Future Work

In the coming months, we intend to extend our metamodel of Frames within XMT to provide support for selection and iteration based on complex OCL-based expressions and to generalise the view of frames such that framed components may comprise UML metamodel elements. We will define and capture a concrete syntax for frames in the tool to obviate the current need to draw frame instances as time-consuming snapshots. Our metamodel of Features will be migrated to XMT and integrated with the meta-frames to investigate the generation of custom artefacts from PLA models. We hope this will lead to a precise and executable specification of both features and frames that will be 'open' to the extent that any suitably equipped meta-modelling tool will be able to implement our specification. By using feature modelling we will provide a view of the individual members of the product family, and by using Frame Technology we will be able to generate and implement configurations and products.

8 Acknowledgements

We gratefully acknowledge the support of BAE Systems with the preparation of the work described herein.

9 References

1. Clark, A., Evans, A., Kent, S., *'A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modelling Approach'*, version 1.0, September 2000

2. Holmes, C., Evans, A., Sammut, P., *'Support for Feature Modelling in the UML'*, BAE-WSC-RP-GEN-020092, Version 1

3. Intermetrics Inc., *'Ada95 Reference Manual – The Language Standard Libraries'*, International Standard, ANSI/ISO/IEC-8652:1995, January 1995

4. Clarke, T., Evans, A., Kent, S., *'A Programmers Guide to MMT'*, copy provided by authors

5. *'A UML Based Specification Environment'*, http://dustbin.informatik.uni-bremen.de/projects/USE (June 2001)

6. *'XMT User Guide'*, pre-release Version 0.1, Xactium, April 2003

7. Kang, K., Cohen, S., Hess, J., Novak, W., Spencer Peterson, A., *'Feature Oriented Domain Analysis (FODA) Feasibility Study'*, CMU/SEI-90-TR-21, ESD-90-TR-222, 1990

8. Warmer, J., Kleppe, A., *'The Object Constraint Language – Precise Modeling with UML'*, Addison-Wesley, 1999

9. Bassett, P.G., *'Framing Software Reuse – Lessons from the Real World'*, Prentice Hall, 1997

10. Jacobson, I., Gris, M., Jonsson, P., *'Software Reuse – Architecture, Process and Organization for Business Success'*, Addison Wesley, 1997

11. Biggerstaff, T., *'The Library Scaling Problem and the Limits of Concrete Component Reuse'*, Int. Conf. On Software Reuse, Rio De Janiero, 1994, pp.102~109

12. Lee, K., Kang, K., Lee, J., *'Concepts and Guidelines of Feature Modeling for Product Line Software Engineering'*, Software Reuse: Methods, Techniques, and Tools, 7th International Conference, ICSR-7, Austin, Texas, April 2002, Proceedings, pp.62~77

13. Wong, T., W., Jarzabek, S., Swe, S., M., Shen, R., Zhang, H., *'XML Implementation of Frame Processor'*, Proceedings of ACM Symposium on Software Reusability, SSR-01, Toronto, Canada, May 2001, pp.164~172

14. Liskov, B., *'Data Abstraction and Hierarchy'*, SIGPLAN-Notices 23, no. 5, May 1988, pp.17~34

15. Rumbaugh, J., Jacobson, I., Booch, G., *'The Unified Modeling Language Reference Manual'*, Addison-Wesley, 1999

16. Guthrey, S., *'Are the Emperor's new clothes object oriented?'*, Dr. Dobb's Journal, December 1989, pp.80~86

17. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stahl, M., *'A System of Patterns – Pattern-Oriented Software Architecture'*, Wiley, 1996

18. Levine, J., Mason, T., Brown, D., *'lex & yacc'*, O'Reily, 1995

19. Ray, E.T., *'Learning XML'*, O'Reilly, 2001

20. Mangano, S., *'XSLT Cookbook'*, O'Reilly, 2002

21. Sauer, F., *'Metadata driven multi-artifact generation using Frame Oriented Programming'*, Position Paper, OOPSLA 2002 Workshop on Generative Techniques in the context of Model Driven Architecture, November 4~8, 2002

22. Völter, M., *'A Catalog of Patterns for Program Generation'*, EuroPloP2003, Eighth European Conference on Pattern Languages of Programs, 25th–29th June 2003, Irsee, Germany, http://hillside.net/europlop/papers/WorkshopB/VoelterM_1.pdf

23. McCarthy, B., *'Weapon of Choice'*, The IEE Review, April 2003, p.34

24. Czarnecki, K., Eisenecker, U., *'Generative Programming – Methods, Tools, and Applications'*, Addison-Wesley 2000

25. Brown, W.E., *'Applied Template Metaprogramming in SI Units: the Library of Unit-Based Computation'*, Computational Physics Department, Fermi National Accelerator Laboratory, Batavia, Illinois, August 23, 2001

26. Landry, W., *'Implementing a High Performance Tensor Library'*, 2nd Workshop on C++ Template Programming, OOPSLA 2001, Florida, October 14, pp.1~11

27. Cleaveland, J.C., *'Program Generators with XML and Java'*, Prentice Hall, 2001

28. Xerox Corporation, *'Aspect Oriented Programming: Going Beyond Objects for Better Separation of Concerns in Design and Implementation'*, presentation material, 1998

29. Wadge, W., *'Intensional Markup Language'*, 3rd International Workshop DCW 2000, Quebec City, 19~21 June, LNCS 1830, Springer-Verlag, pp.82~89

30. Plaice, J., Wadge, W., *'A New Approach to Version Control'*, IEEE Transactions on Software Engineering, Vol. 19, No. 3, March 1993, pp.268~276

31. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *'Design Patterns – Elements of Reusable Object-Oriented Software'*, Addison-Wesley, 1995

32. Jarzabek, S., Ong, W., C., Zhang, H., *'Handling Variant Requirements in Domain Modeling'*, Proceedings of 13th International. Conference on Software Engineering & Knowledge Engineering, SEKE'01, Knowledge Systems Institute, June 2001, Buenos Aires, Argentina

33. Patzke, T., Muthig, D., *'Product Line Implementation with Frame Technology: A Case Study'*, IESE-Report No. 018.03/E, Version 1.0, March 2003, Fraunhofer Institut Experimentalles Software Engineering

34. Audsley, N., Bate, I., Crook-Dawkins, S., *'Automatic Code Generation for Airborne Systems'*, Proceedings of the IEEE Aerospace Conference, March 8~15, 2003

35. McIlroy, M., D., *'Mass-Produced Software Components'*, NATO Science Committee, Garmisch, Germany, 7~11 October 1968

36. Jones, C. A., Drake, F. L., *'Python & XML'*, O'Reilly, 2002

37. Tilbrook, D., Crook, R., *'Large Scale Porting through Parameterization'*, USENIX Association, Proceedings of the Summer 1992 USENIX Conference, 8~12-June 1992, San Antonio, Texas, USA, pp.209~216