Feature Oriented Model Driven Development: A Case Study for Portlets

Salvador Trujillo
Department of Computer Sciences
University of the Basque Country
20009 San Sebastian, Spain
struji@ehu.es

Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
batory@cs.utexas.edu

Oscar Diaz
Department of Computer Sciences
University of the Basque Country
20009 San Sebastian, Spain
oscar.diaz@ehu.es

Abstract

Model Driven Development (MDD) is an emerging paradigm for software construction that uses models to specify programs, and model transformations to synthesize executables. Feature Oriented Programming (FOP) is a paradigm for software product lines where programs are synthesized by composing features. Feature Oriented Model Driven Development (FOMDD) is a blend of FOP and MDD that shows how products in a software product line can be synthesized in an MDD way by composing features to create models, and then transforming these models into executables. We present a case study of FOMDD on a product line of portlets, which are components of web portals. We reveal mathematical properties of portlet synthesis that helped us to validate the correctness of our abstractions, tools, and specifications, as well as optimize portlet synthesis.

1 Introduction

Model Driven Development (MDD) is an emerging paradigm for software development that specifies programs in *domain-specific languages (DSLs)*, encourages greater degrees of automation, and exploits standards [10][11][24]. MDD uses models to represent a program. A *model* is written in a DSL that specifies particular details of a program's design. As an individual model captures limited information, a program is often specified by several different models. A model can be derived from other models by transformations, and program synthesis is the process of transforming high-level models into executables (which are also models).

Feature Oriented Programming (FOP) is a paradigm for software product lines where programs are synthesized by composing features [6]. A *feature* is an increment of program functionality, and is implemented by refinements that extend existing artifacts and that add new artifacts (code, makefiles, documentation, etc.). When features are composed, consistent artifacts that define a program are synthesized. A tenet of FOP is the use of algebraic techniques to specify and manipulate program designs.

Feature Oriented Model Driven Development (FOMDD) is a blend of FOP and MDD. Models can be *refined* by composing features (a.k.a. *endogenous transformations* that map models expressed in the same DSL [30]), and can be *derived* from other models (a.k.a. *exogenous transformations* that map models written in different DSLs [30]).

We present a case study of FOMDD that is a product-line of portlets, which are building blocks of web portals. We explain how we specify a portlet as a set of models from which we refine and derive an implementation. Combining model *derivation* and model *refinement* exposes a fundamental commuting relationship that should arise in all examples of FOMDD: namely, the transformation of a composed model equals the composition of transformed models. Hence, an executable can be synthesized in very different ways. Commuting relationships impose stringent properties on domain models and the implementations; they have helped us validate the correctness of our abstractions, tools, and portlet specifications, as well as optimize portlet synthesis. We begin with a review of MDD, FOP, and portlets.

2 Background

2.1 Model Driven Development

Program specification in MDD uses one or more *models* to define a target program. Ideally, these models are *platform independent (PIM)*. Model *derivations* convert platform independent models to *platform specific models (PSM)*, where assorted technology bindings are introduced. Possible results of transforming PIMs can be an executable or an input to an analysis tool, where both an executable and an analysis-input file are themselves considered models.

Metaprogramming is the concept that program development is a computation. We assert that MDD is a metaprogramming paradigm [8]. That is, models are values and transformations are functions that map values to other values. Scripts that transform models into executables are *metaprograms* (i.e., programs that manipulate values that themselves are programs). For example, `ant` makefiles are metaprograms; the values of a makefile are files (programs) and the execution of a makefile can produce an executable [2]. An MDD process can be written as a makefile (metaprogram) whose input values are DSL specifications (models) of target programs, and whose output values are synthesis targets. We will see examples of such metaprograms in Section 3.

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

2.2 Feature Oriented Programming

FOP is a paradigm for creating software product lines [6]. Features (a.k.a. feature modules) are the building blocks of programs. An *FOP model* of a product line is an algebra that offers a set of operations, where each operation implements a feature. We write `M = {f, h, i, j}` to mean model `M` has operations or features `f`, `h`, `i`, and `j`. FOP distinguishes features as *constants* or *functions*. Constants represent base programs. For example:

```
f       // a base program with feature f
h       // a base program with feature h
```

Functions represent *program refinements* that extend a program that is received as input. For instance:

```
i•x     // adds feature i to program x
j•x     // adds feature j to program x
```

where • denotes function application.

The design of a program is a named expression, e.g.:

```
prog1 = i•f     // prog1 has features f and i
prog2 = i•j•h   // prog2 has features h,j,i
```

The set of programs that can be created from an FOP model is its product line. Expression optimization corresponds to program design optimization, and expression evaluation corresponds to program synthesis [4][39].[1]

> Note: Although we write the composition of features `a` and `b` as `a•b`, it really is an abbreviation of the expression `compose(a,b)`. We use • to simplify FOP expressions.

The connection of FOP to metaprogramming and MDD is simple: FOP treats programs as values, and features are functions that map values to other values. In Section 4, we show how FOP and MDD can be integrated.

2.3 Portlets

A *portal* is a web page that provides centralized access to a variety of services [16]. An increasing number of these services are not offered by the portal itself, but by a third-party component called a *portlet*. Figure 1 depicts a 3-tier architecture for portlets, where an end-user's `MyBrowser` accesses the `MyPortal` page through HTTP. `MyPortal` is hosted by `Consumer1` and consists of a layout aggregating (through SOAP [44]) the `Alpha`, `Beta`, and `Delta` portlets that are hosted by different producers.

Unlike web services, which offer only business logic methods, portlets additionally provide a presentation-oriented web service. Hence, portlets not only return raw data but also renderable markup (e.g. XHTML) that can be displayed within a portal page. Figure 2 shows the MyYahoo portal [46], where a variety of services are provided, some of which may be portlets.

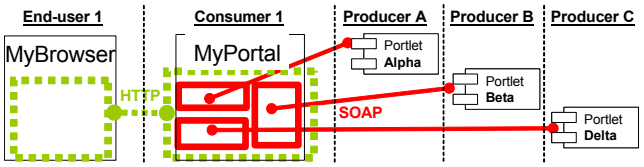

Figure 1. A 3-Tier Architecture for Portlets

Figure 2. MyYahoo Portal

Until recently, portlet realization was dependent on the infrastructure of the producer of portlets (service provider) and the portal consumer (service container). This changed with the release of the *Web Services for Remote Portlets (WSRP)* [32], which standardized the web service interface between portlet consumers and portlet producers, and the *Java Specification Request 168 (JSR 168)* [23], which defined how portlets should be implemented in Java. These standards foster a COTS market, where portlets can be deployed independently of the platform on which they were developed. Further, different customers demand different portlets that overlap in functionality. Consequently, techniques for customizing portlets are increasingly sought [17][42].

Our experience with portlet implementations is that a sizable fraction of their code is common. This led us to create an OO framework (using eXo portal platform [18]) that is realized by 85 classes and 9 KLOC Java. It encapsulates and reuses logic and infrastructure common to all portlets and provides the base functionality on top of which application-specific functionality is built. We created a *Domain Specific Language for Portlets (PSL)* to define this functionality. A PSL specification is represented by several XML documents. The next section shows how portlets can be synthesized using MDD.

1. The use of one feature may preclude the use of some features or may demand the use of others. Tools that validate compositions of features are discussed in [7].

3 Model Driven Development of Portlets

Portlet MDD (PMDD) is a model-driven approach that automates portlet implementation. *PinkCreek* is a portlet that provides flight reservation capabilities to different portals. Its functionality is roughly: (i) search for flights, (ii) present flight options, (iii) select flights, and (iv) purchase tickets. In this section, we illustrate PMDD using *PinkCreek* as an example, and explain its MDD metaprogram.

3.1 Step 1: Define Portlet Controller

A State Chart (SC) provides a platform independent model for representing the flow of computations in a portlet [19][22][33]. Each portlet consists of a sequence of states where each state represents a portlet page. States are connected by transitions whose handlers either execute some action, render some view, or both.

Figure 3. SC fragment from *PinkCreek*

Figure 3 shows an SC diagram fragment for *PinkCreek* where each state represents a step in making a flight reservation. We removed transition details to make the figure clearer. Existing tools (e.g. IBM Rational Rose, Poseidon) can draw an SC model in UML notation, and represent the model as an XMI specification [34]. We specify state charts in the W3C SCXML language [43]. Figure 4 lists a fragment of a *PinkCreek* specification.

```
<scxml version="1.0" initialstate="s1Search">
    <state id="s1Search">
        <transition event="actionEvent">
            <target next="s1Search"/>
            <action>LoadAirport</action>
        </transition>
        <transition event="viewEvent">
            <target next="s2Select"/>
            <action>SearchFlightView</action>
        </transition>
    </state>
    <!-- remaining states omitted -->
</scxml>
```

Figure 4. SC.xml fragment for *PinkCreek*

The SC of a portlet defines its controller; the details of actions and views are defined elsewhere. So the first step is to define the SC of the portlet's controller. The next step is to map an SC specification to a PSL specification.

3.2 Step 2: Map SC to PSL

A PSL specification of a portlet consists of three distinct XML documents. One document ($\texttt{PSL}_{\texttt{ctrl}}$) defines a state machine controller for a portlet. The other two documents ($\texttt{PSL}_{\texttt{act}}$, $\texttt{PSL}_{\texttt{view}}$) define the actions to be performed and the views to be rendered during controller execution. The production of each is described next.

3.2.1 Step 2.1: Transforming SC to PSL_{ctrl}

We designed the PSL controller language prior to the release of the SCXML standard. As we now use the standard, we reused our PSL language and framework by writing an XSL translator ($\texttt{T}_{\texttt{sc2ctrl}}$) that maps a state chart specification `SC` to a PSL controller document ($\texttt{PSL}_{\texttt{ctrl}}$):

$$\texttt{PSL}_{\texttt{ctrl}} = \texttt{T}_{\texttt{sc2ctrl}}\ (\texttt{SC}) \qquad (1)$$

The $\texttt{PSL}_{\texttt{ctrl}}$ document is interpreted by our Portlet framework at portlet execution time.

3.2.2 Step 2.2: Transforming PSL_{ctrl} to PSL_{act}

Activities in a state chart are defined as actions to perform and views to render when a transition occurs. An *action skeleton* is an interface that defines only the names of action methods. (We will see shortly that there is a corresponding view skeleton that defines the names of view methods). An action skeleton ($\texttt{PSL}_{\texttt{act-sk}}$) is derived by a simple analysis of the $\texttt{PSL}_{\texttt{ctrl}}$ document that extracts action names. The transformation ($\texttt{T}_{\texttt{ctrl2act}}$) implements this derivation:

$$\texttt{PSL}_{\texttt{act-sk}} = \texttt{T}_{\texttt{ctrl2act}}\ (\texttt{PSL}_{\texttt{ctrl}}) \qquad (2)$$

Figure 5 shows a snippet of $\texttt{PSL}_{\texttt{act-sk}}$ skeleton, where the name of action `pinkcreek.LoadAirport` was extracted.

```
<Portlet.actions id="PinkCreek">
    <Action id="LoadAirport" type="JAVA_CLASS">
        <class>pinkcreek.LoadAirport</class>
        <!-- TODO Params -->
        <!-- TODO Results -->
    </Action>
    <!-- remaining actions omitted -->
</Portlet.actions>
```

Figure 5. PSL_{act}.xml skeleton ($\texttt{PSL}_{\texttt{act-sk}}$) for *PinkCreek*

The name of an action is not sufficient: we still need to specify the input (`Params`) and output (`Results`) of each action method. This platform-specific information is added by refining the generated $\texttt{PSL}_{\texttt{act-sk}}$ document. Figure 6 shows a snippet of such a refinement (denoted $\Delta\texttt{PSL}_{\texttt{act-usr}}$) that extends the `pinkcreek.LoadAirport` method with its input parameters and output result.[2] The tool that we use to compose an XML file with its refinements is described in [41].

2. $\Delta\texttt{PSL}_{\texttt{act-usr}}$ (and its counterpart $\Delta\texttt{PSL}_{\texttt{view-usr}}$) are platform-specific, as the parameters to actions and views are not platform invariant. Alternatively, some MDD approaches define code in a platform-independent language and translate code to a platform-specific language [24].

```
<xak:refines id="PinkCreek">
    <xak:extends id="LoadAirport">
        <xak:super id="LoadAirport"/>
        <Param name="from" value="$data/orgn"/>
        <Param name="to" value="$data/dest"/>
        <!-- remaining Params omitted -->
        <Result name="flight"/>
        <!-- remaining Results omitted -->
    </xak:extends>
</xak:refines>
```

Figure 6. PSL_{act}.xml user refinement ($\Delta PSL_{act\text{-}usr}$) fragment for *PinkCreek*

Composing the generated action skeleton ($PSL_{act\text{-}sk}$) with its user hand-written refinement ($\Delta PSL_{act\text{-}usr}$) yields a complete PSL action document (PSL_{act}), which defines the name, type, and parameters of each action method:

$$PSL_{act} = \Delta PSL_{act\text{-}usr} \bullet PSL_{act\text{-}sk} \quad (3)$$

3.2.3 Step 2.3: Transforming PSL_{ctrl} to PSL_{view}

An identical procedure is used to create a PSL view document (PSL_{view}) from the PSL controller document (PSL_{ctrl}). A view skeleton ($PSL_{view\text{-}sk}$) is generated from PSL_{ctrl}, and it is composed with a hand-written refinement ($\Delta PSL_{view\text{-}usr}$) that refines view methods with their input parameters, to yield the desired view document[3]:

$$PSL_{view\text{-}sk} = T_{ctrl2view}\ (PSL_{ctrl}) \quad (4)$$
$$PSL_{view} = \Delta PSL_{view\text{-}usr} \bullet PSL_{view\text{-}sk} \quad (5)$$

At this point, we have a PSL specification for a portlet. However, additional platform-specific implementation details remain to be given.

3.3 Step 3: from PSL to Implementation

A PSL specification almost completely defines what the interpreter needs to execute a portlet. What is lacking is (i) business logic of each action method, and (ii) the logic to draw the layout page of each view method.

3.3.1 Step 3.1: Transforming PSL_{act} to Jak Code

Jak(arta) is a superset of the Java language, where class and method refinements can be declared [6]. Jak is the primary language for implementing refinements in FOP.

PSL_{act} is an XML document that sketches the source code skeleton of a set of Jak classes: it specifies the signatures of all portlet-specific methods. We can generate skeletal Jak classes (Jak_{sk}) by a transformation ($T_{act2jak}$) of PSL_{act}. A unique Jak class is generated for each action in PSL_{act}. Also generated are portlet-specific methods and members that are required by our portlet framework.

$$Jak_{sk} = T_{act2jak}\ (PSL_{act}) \quad (6)$$

3. View methods return no results.

```
import java.util.Hashtable;
import org.onekin.pf.action.jc.IJavaAction;
public class LoadAirport implements IJavaAction
{
 /** This is the default Constructor ... */
 public LoadAirport()
 { /* empty */ }

 /** This method executes ... */
 public void execute(Hashtable prm,Hashtable rs)
 { /* empty */ }
}
```

Figure 7. `LoadAirport.jak` fragment for *PinkCreek*

Figure 7 shows the derived Jak code for the action `pinkcreek.LoadAirport`. Note that extra methods (e.g., `execute`) that are specific to our portlet framework are also produced, along with additional data members (not shown). Their generation simplifies the development of user code provided in the next step.

Jak code is generated instead of Java because the actions of the generated methods must be completed by a programmer. We complete the generated skeleton (Jak_{sk}) by composing it with a hand-written refinement (ΔJak_{usr}) that encapsulates the business logic for each method:

$$Jak_{code} = \Delta Jak_{usr} \bullet Jak_{sk} \quad (7)$$

Figure 8 shows the corresponding refinement for Figure 7.

```
refines class LoadAirport {
 public LoadAirport()
 { /* USER CODE GOES HERE */ }

 public void execute(Hashtable prm,Hashtable rs)
 { /* USER CODE GOES HERE */ }
}
```

Figure 8. `LoadAirport.jak` refinement for *PinkCreek*

3.3.2 Step 3.2: Transforming PSL_{view} to JSP

In an analogous manner, `JSP` code skeletons (Jsp_{sk}) are created from PSL_{view}, one JSP page per view. Each skeleton is completed by composing it with a hand-written refinement (ΔJsp_{usr}). The result is a compilable set of JSP files (Jsp_{code}), one per PSL_{view} view method:

$$Jsp_{sk} = T_{view2jsp}\ (PSL_{view}) \quad (8)$$
$$Jsp_{code} = \Delta Jsp_{usr} \bullet Jsp_{sk} \quad (9)$$

3.4 Step 4: Building the Product

A PSL specification together with Jak_{code} and Jsp_{code} form the *raw material* of a Portlet product (P_{raw}). Other artifacts ($\Delta P_{additional}$) are needed, such as deployment descriptors and JAR libraries, to complete a portlet's source (P_{src}). These artifacts are introductions (i.e., new artifacts that do not refine existing ones) that are added by composing $\Delta P_{additional}$ to P_{raw}:

$$P_{raw} = \{PSL_{ctrl}, PSL_{act}, PSL_{view}, Jak_{code}, Jsp_{code}\} \quad (10)$$

$$P_{src} = \Delta P_{additional} \bullet P_{raw} \quad (11)$$

Among the artifacts added by $\Delta P_{additional}$ is an `ant` makefile [40], which builds the *web archive (WAR)* of the portlet. Executing the makefile translates Jak files to Java files, compiles Java files into class files, and creates the portlet web archive (P_{war}) which is deployed into the target portal.

$$P_{war} = antBuild\ (P_{src}) \quad (12)$$

3.5 Recap and Perspective

Our PMDD process for building *PinkCreek* is a straightforward metaprogram. Given all the inputs (i.e., MDD models) that define a portlet, namely (SC, $\Delta PSL_{act\text{-}usr}$, $\Delta PSL_{view\text{-}usr}$, ΔJak_{usr}, ΔJsp_{usr}), the process automatically generates the portlet's WAR (P_{war}). Model refinements are expressed as endogenous transformations, and model derivations are exogenous transformations [30]. Figure 9 shows this process as three functions (T_{mkraw}, $T_{raw2war}$, T_{sc2war}). (The reason why we used three functions, instead of one, will become clear in Section 4). T_{sc2raw} automates significant and tedious tasks in portlet development. For example, 59 files and 4250 LOC are derived from an input of 10 files and 730 LOC.

```
Tmkraw(SC,ΔPSLact-usr,ΔPSLview-usr,ΔJakusr,ΔJspusr)
{   PSLctrl = Tsc2ctrl (SC);                  // (1)
    PSLact-sk = Tctrl2act (PSLctrl);          // (2)
    PSLact = ΔPSLact-usr • PSLact-sk;         // (3)
    PSLview-sk = Tctrl2view (PSLctrl);        // (4)
    PSLview = ΔPSLview-usr • PSLview-sk;      // (5)
    Jaksk = Tact2jak (PSLact);                // (6)
    Jakcode = ΔJakusr • Jaksk;                // (7)
    Jspsk = Tview2jsp (PSLview);              // (8)
    Jspcode = ΔJspusr • Jspsk;                // (9)
    Praw = { PSLctrl, PSLact, PSLview,
             Jakcode, Jspcode };              // (10)
    return Praw; }
Traw2war(Praw)
{   Psrc = ΔPadditional • Praw                // (11)
    Pwar = antBuild (Psrc)                    // (12)
    return Pwar; }
Tsc2war(SC,ΔPSLact-usr,ΔPSLview-usr,ΔJakusr,ΔJspusr)
{   Praw = Tmkraw(SC,ΔPSLact-usr,ΔPSLview-usr,
             ΔJakusr,ΔJspusr);
    return Traw2war(Praw); }
```

Figure 9. Metaprograms for PMDD

Of the five inputs that we need to specify, only one (the statechart) is platform-independent. The remaining are platform-specific, expressing customized business logic and view logic. Ideally, these remaining inputs should be derived from one or more PIMs, which would marginally alter the metaprogram of Figure 9. Although we do not yet have such PIMs, this does not impact the results of this paper. In general, our situation is symptomatic of a general problem in MDD on how to express customized business logic in PIMs. It is common to use model "escapes" from which code can be specified. Sometimes a generic programming language is used to express code fragments, from which Java or C# is produced [24]. Creating declarative models for all PMDD inputs seems unlikely.

4 Feature Oriented MDD

The input to our PMDD process is a 5-tuple <SC,$\Delta PSL_{act\text{-}usr}$,$\Delta PSL_{view\text{-}usr}$,$\Delta Jak_{usr}$,$\Delta Jsp_{usr}$>, which we abbreviate as <s,a,v,b,j>. Given a tuple that defines a portlet, the transformation T_{sc2war} synthesizes the portlet's WAR file.

Portlets are like other software applications: there is a family of related portlet designs and capabilities that we want to create. The designs and capabilities that differentiate one portlet from another can be explained in terms of features (i.e., increments in portlet functionality). Instead of manually creating portlet specifications (5-tuples), we want to synthesize their 5-tuples using FOP.

An FOP model of a portlet domain includes one or more base portlets called *constants*, and one or more refinements, called *functions*, that add functionality to a portlet (Section 2.2). A constant `C` is a 5-tuple <s,a,v,b,j>. A function is also a 5-tuple <Δs,Δa,Δv,Δb,Δj> that defines changes to a base tuple in terms of:

- refinements to a base ***s***tate chart (Δs)
- refinements to a base ***a***ction document (Δa)
- refinements to a base ***v***iew document (Δv)
- refinements to a base actions ***b***usiness logic (Δb)
- refinements to a base ***j***sp page (Δj)

Suppose we want to synthesize the 5-tuple <s′,a′,v′,b′,j′> of a portlet `P` by starting with a base portlet `C` (a constant) and refining it by the features (functions) `F1` and `F2`. Our portlet specification `P` is:

$$\begin{aligned} P &= F2 \bullet F1 \bullet C \\ &= \langle \Delta s_2, \Delta a_2, \Delta v_2, \Delta b_2, \Delta j_2 \rangle \bullet \\ &\quad \langle \Delta s_1, \Delta a_1, \Delta v_1, \Delta b_1, \Delta j_1 \rangle \bullet \langle s, a, v, b, j \rangle \\ &= \langle \Delta s_2 \bullet \Delta s_1 \bullet s,\ \Delta a_2 \bullet \Delta a_1 \bullet a,\ \Delta v_2 \bullet \Delta v_1 \bullet v,\ \Delta b_2 \bullet \Delta b_1 \bullet b, \\ &\quad \Delta j_2 \bullet \Delta j_1 \bullet j \rangle \\ &= \langle s', a', v', b', j' \rangle \end{aligned} \quad (13)$$

That is, the 5-tuple of our desired portlet is synthesized by composing the base state chart with its refinements, the base action document with its refinements, and so on [6]. In this section, we explain the interesting challenges we faced in developing portlet features in a model-driven way. It required an extension of PMDD to cope with product lines.

4.1 Developing Feature Constants

A feature constant is developed as a base portlet, as discussed in Section 3. It is defined by a 5-tuple and represents a base portlet to which more features can be added.

4.2 Developing Feature Functions

4.2.1 Challenge 1: Model Refinement

A model in MDD is a specification of a program (or some part or view of a program). Model refinement elaborates a model to reflect the changes made by adding a feature.

A state chart refinement[4] adds new states, new transitions, and refines the activities associated with existing states or transitions [5][28][29]. For example, consider a feature **seat** that extends the *PinkCreek* portlet to allow passengers to select their seat after purchasing flight tickets. Figure 10 shows how a new state **s6Seating** is added to handle the seat selection pages, and how state **s5Itinerary** is refined by linking it with **s6Seating**.

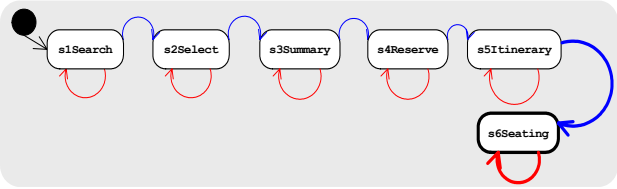

Figure 10. SC Refinement for *PinkCreek seating*

A state chart is defined by an XML document. A refinement of a state chart can also be defined in an XML document (and the tool in [41] can compose such documents).

4.2.2 Challenge 2: Transforming Refinements

Recall that a feature function is a 5-tuple of refinements $\langle \Delta s, \Delta a, \Delta v, \Delta b, \Delta j \rangle$. Defining the changes to a state chart is easy. However, defining the remaining artifact refinements is a bit harder. This section presents the approach that we took to develop feature function 5-tuples.

Recall from Section 3 that P_{raw} is the raw material from which we could build a portlet WAR file. We want to generate a raw material refinement ΔF_{raw} for each feature F that can be added to a base portlet. If we could do so, we could synthesize the raw material for a target portlet. For example, suppose we want to synthesize the raw material for portlet $P{=}F2{\bullet}F1{\bullet}C$ by composing the raw material (C_{raw}) for base feature C and the raw material ($\Delta F1_{raw}$ and $\Delta F2_{raw}$) of its feature functions $F1$ and $F2$. The raw material for portlet P would be:

$$P_{raw} = \Delta F2_{raw} \bullet \Delta F1_{raw} \bullet C_{raw} \quad (14)$$

To accomplish this, we must derive the raw material refinements for each feature function. More precisely, let feature function F be defined by the 5-tuple $\langle \Delta s, \Delta a, \Delta v, \Delta b, \Delta j \rangle$. We want a transformation T'_{mkraw} that maps the 5-tuple of any feature function F to its raw material ΔF_{raw}. The details of the T'_{mkraw} process are given in the Appendix and are virtually identical to the metaprogram of Figure 9 that maps a tuple $\langle s,a,v,b,j \rangle$ to raw materials. Figure 11 shows this process as the metaprogram T'_{mkraw}. For a typical feature, the output size is 3-5 times the size of the input. For the **seat** feature in the *PinkCreek* product line, 27 files (755 LOC) are derived from an input of 9 files and 163 LOC. As in the case of T_{mkraw}, T'_{mkraw} automates significant and tedious tasks in portlet development.

```
T'mkraw(ΔFsc, ΔFact-usr, ΔFview-usr, ΔFjak-usr, ΔFjsp-usr)
{
    ΔFctrl = T'sc2ctrl (ΔFsc);          // (1)
    ΔFact-sk = T'ctrl2act (ΔFctrl);     // (2)
    ΔFview-sk = T'ctrl2view (ΔFctrl);   // (3)
    ΔFact = ΔFact-usr • ΔFact-sk;       // (4)
    ΔFview = ΔFview-usr • ΔFview-sk;    // (5)
    ΔFjak-sk = T'act2jak (ΔFact);       // (6)
    ΔFjsp-sk = T'view2jsp (ΔFview);     // (7)
    ΔFjakcode = ΔFjak-usr • ΔFjak-sk;   // (8)
    ΔFjspcode = ΔFjsp-usr • ΔFjsp-sk;   // (9)
    ΔFraw = {ΔFctrl, ΔFact, ΔFview,
             ΔFjakcode, ΔFjspcode}      // (10)
    return ΔFraw;
}
```

Figure 11. The Metaprogram T'_{mkraw}

4.3 Product Synthesis

Our PinkCreek product line has 26 features (constants and functions), yielding hundreds of interesting and distinct portlets. A particular portlet is specified by an FOP expression that composes a **base** portlet with zero or more extending features (**seat**, **checkin**, etc.):

$$PinkCreek_1 = seat \bullet base$$
$$PinkCreek_2 = checkin \bullet seat \bullet base$$
$$\ldots \text{ // other products}$$

We synthesized portlets by deriving the raw materials of the base and refining features, and composing them. Let the 5-tuples for **base**, **seat**, and **checkin** be $\langle\rangle_{base}$, $\langle\rangle_{seat}$, and $\langle\rangle_{checkin}$. The raw material for $PinkCreek_2$ is computed by:

$$PinkCreek_{raw} = T'_{mkraw}(\langle\rangle_{checkin}) \bullet T'_{mkraw}(\langle\rangle_{seat}) \bullet T_{mkraw}(\langle\rangle_{base}) \quad (15)$$

Given the raw material of a portlet, we invoke the $T_{raw2war}$ transformation of Figure 9 to produce the portlet's WAR:

$$PinkCreek_{war} = T_{raw2war}(\ PinkCreek_{raw}\ )$$

4. Note that a refinement is based on mixin inheritance. Weber et al. extend statecharts by regular inheritance [45].

5 Commuting Diagrams

Synthesizing portlets by composing raw materials is *not* the way we originally planned. Our intent in Section 4 was to synthesize the 5-tuple of a portlet by composing the 5-tuples of its base and refining features, such as:

$$\langle\rangle_{PinkCreek2} = \langle\rangle_{checkin} \bullet \langle\rangle_{seat} \bullet \langle\rangle_{base}$$

And use T_{mkraw} of Figure 9 to derive portlet raw material:

$$PinkCreek_{raw} = T_{mkraw}(\langle\rangle_{PinkCreek2}) \quad (16)$$

We now had *two* different ways to produce the raw materials of a portlet, namely (15) and (16). That is, a transformation of a composition of 5-tuples (16) equals the composition of the transformation of each 5-tuple (15):

$$T_{mkraw}(\langle\rangle_{F1} \bullet \langle\rangle_{base}) = T'_{mkraw}(\langle\rangle_{F1}) \bullet T_{mkraw}(\langle\rangle_{base}) \quad (17)$$

Figure 12 illustrates (17): we synthesized portlets via the path labeled A (15), but had an alternate path B (16).

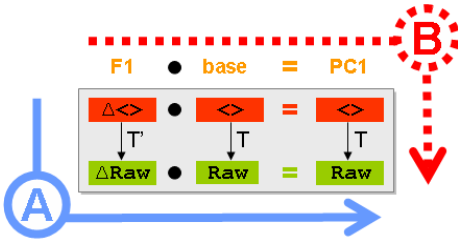

Figure 12. Alternative Synthesis Paths

FOMDD explicitly combines model refinement and model derivation. Our research exposed a fundamental relationship between the two, which is expressed in Figure 13 as a *commuting diagram* [37], where M_0, M_1, D_0, and D_1 are domains and $\Delta M: M_0 \rightarrow M_1$, $\Delta D: D_0 \rightarrow D_1$ and $f: (M_0 \cup M_1) \rightarrow (D_0 \cup D_1)$ are functions satisfying:

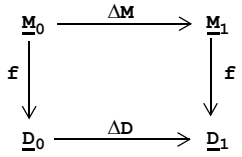

Figure 13. Commuting Diagram

$$f \bullet \Delta M = \Delta D \bullet f \quad (18)$$

In PMDD, we encountered instances of these domains: $M_0 \in M_0$, $M_1 \in M_1$, $D_0 \in D_0$, and $D_1 \in D_1$. We refined model M_0 by ΔM to produce model M_1. Function or transformation f derived model D_1 from M_1. Alternatively, we could derive model D_0 from M_0 using function f, and then refine D_0 by ΔD (that corresponds to ΔM) to produce D_1. An operator f' maps function ΔM to function ΔD. The general relationship is:

$$f(\Delta M \bullet M_0) = f'(\Delta M) \bullet f(M_0) \quad (19)$$

where (17) is a PMDD instance of (19) which in our case states that the transformation of a composed model equals the composition of transformed models. Note that no special restrictions are placed on models and features by commuting diagrams, except that (19) must hold.

The reason why (17) holds is because functions T_{mkraw} and T'_{mkraw} are *morphisms* (i.e., structure preserving mappings [37]). Formally proving structure preservation is difficult, as it requires a formalization of the input and output domains, a formalization of the properties to be preserved, and a faithful implementation of this formalization, each step of which is a non-trivial undertaking. An alternative approach is to validate each *instance* of a transformation. For example, Narayanan and Karsai [31] presented an algorithm to validate that the translation between two different state chart representations preserves each state, transition, and activity (and no additional states, transitions, and activities are added). This is accomplished by maintaining an internal mapping between input and output representations and validating that there is a 1-1 correspondence between input/output states, transitions, and activities.

We took a different approach by computing the results in *both* directions (paths A and B) and used a source equivalence "diff" to test for equality. *Source equivalence* is syntactic equivalence with two relaxations: it allows permutations of members when member ordering is not significant and it allows white space to differ when white space is unimportant. We added this extra computation as an option to our metaprograms to validate (17).

We soon discovered that there are *many* other commuting diagrams/relationships in PinkCreek. For example, state charts can be refined and then mapped to PSL controllers, or a PSL controller can be derived from a state chart and then refined:

$$T_{sc2ctrl}(\Delta F_{SC} \bullet B_{SC}) = T'_{sc2ctrl}(\Delta F_{SC}) \bullet T_{sc2ctrl}(B_{SC}) \quad (20)$$

where B_{SC} is a base state chart and ΔF_{SC} is a state chart refinement of feature F. These relationships helped us validate individual transformations of Figure 9 and Figure 11.

5.1 Experience

Initially, our tools did not satisfy (17). That is, synthesizing portlet raw material via paths A and B yielded different results. Upon closer inspection, we discovered errors in both our tools and portlet specifications. Such errors were not exposed until we synthesized raw materials via path B.

We soon realized the significance of commuting diagrams/relationships. While checking validity increases build times (more in Section 5.2), *we obtain assurances on the correctness of our PMDD abstractions, our portlet specifications, and our tools.* (17) defines stringent properties that our models, tools, and specifications must satisfy, *and without*

these diagrams, we were unaware that these constraints existed. Our results are general: their benefits will hold in the development of tools, models, and specifications for other domains using FOMDD, as they too will have commuting diagrams like Figure 13. More on this in Section 6.

5.2 Build Optimization

Figure 12 offers two ways in which portlet raw materials can be synthesized: either build raw materials via path A or via path B. Figure 14 shows experimental results of synthesizing portlets via each path. The A line indicates the cost of traversing path A, which includes the cost of transforming 5-tuples to raw materials. Tuple transformations only have to be computed once, as raw materials of model refinements are portlet invariant. This offers a very useful optimization: raw materials only have to be computed once. The A-T line shows the reduction in cost by this optimization. Note that the A-T path is 2-3 times faster than path B (indicated by line B). The results confirm our intuition: composing features and transforming (path B) is substantially faster than transforming and then composing features (path A). However, when the raw material optimization is used, the reverse is true (path A-T is faster than B). If we validate compositions by building both ways (A+B-T), build times increase, but this is a one-time cost.

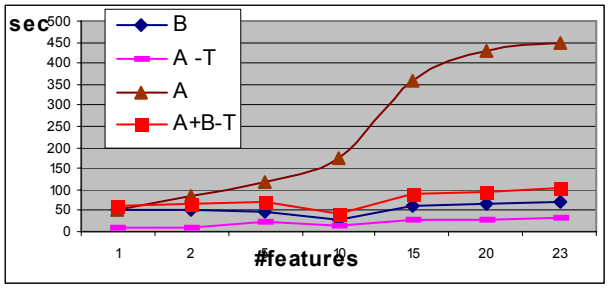

Figure 14. *PinkCreek* Build Time Alternatives

In general, metrics other than cost might be used in path selection (e.g., ease of tooling).

6 Related Work

Model derivation and model refinement are common in FOMDD. We expressed derivations by exogenous transformations (mappings of models written in different DSLs) and refinements by endogenous transformations (mappings of models written in the same DSL). We explicitly represent MDD processes as functional metaprograms (where programs are values and transformations are functions that map programs); this idea is latent in the MDD community. We took an additional step forward by merging MDD ideas with those of FOP, which itself has a long history of development [4]-[8]. A complimentary view which describes MDD and FOP as an object-oriented metaprogramming paradigm is given in [8].

Horizontal and vertical transformations are also common in MDD [30]. Horizontal transformations map source and target models at the same level of abstraction (e.g, refactorings), while vertical transformations map models that reside at different levels of abstraction (PIM to PSM mappings). We have clearly used both kinds of transformations in PinkCreek, but we found no advantage in making horizontal and vertical distinctions in our work.

Much of the tooling effort in MDD today is focussed on UML models. What is generally lacking are tools to express *refinements* of UML models, on which FOMDD relies. Building such tools is the subject of future work.

Kurtev uses XML transformations to develop XML applications [26]. The design of web applications includes not only functionality but also content, navigation and presentation issues. This calls for a model-based approach (e.g. W2000 [3], WebML [12], UWE [25] or OO-HMethod [35]) from which web applications are derived [36]. However, we are unaware of MDD approaches for building portlets.

Merging MDD and product lines is not new [1][13][14][15][20][21][38], we know of few examples that explicitly use features in MDD. One is *BoldStroke*: a product-line written in several millions lines of C++ for supporting a family of mission computing avionics for military aircraft. Gray used MDD to express maintenance tasks on BoldStroke [21]. Adding a feature required both updating BoldStroke's model and code base. Although build optimizations were used (e.g., delaying the updates of the code base after model refinement), no commuting relationships were used, although we believe they exist.

Proving properties of large programs remains a difficult challenge. The programs used in PMDD (`javac`, `XSLT`, AHEAD tools) may be on a scale that is appropriate for the Verified Software Grand Challenge of Hoare, Misra, and Shankar [27], which seeks scalable technologies for program verification.

We mentioned earlier that commuting diagrams are common in FOMDD. In the construction of the AHEAD tool suite, for example, customized parsers were built by first composing a base grammar with its refinements, and then using `JavaCC` to derive a parser [6]. This is comparable to "path B" in Figure 12. A counterpart to path A would be to compose a base parser with its refinements. Unfortunately, `JavaCC` translates only complete grammars into complete parsers (not grammar refinements into parser refinements), so path A could not be evaluated.

Commuting diagrams are fundamental to *category theory (CT)* [37], which is a general mathematical theory of structures and of systems of structures. A benefit of FOMDD is that it is mathematically based, and this makes connections with category theory easier to recognize. PinkCreek has provided us with an invaluable example that has enabled us to unify the ideas of FOMDD program synthesis and CT. An exposition of these ideas is the subject of forthcoming work [9].

7 Conclusions

MDD and FOP are complementary paradigms. MDD derives models and FOP refines models. Metaprogramming unifies models with values; transformations map values to values. This unification of MDD and FOP, here called FOMDD, offers a powerful paradigm for creating product lines using MDD technology.

We presented a case study of FOMDD that created a product line of portlets. We showed how the MDD production of a portlet is a metaprogram that transforms a multi-model specification of a portlet into a web archive file. We expressed variations in portlet functionality as features, and synthesized portlet specifications by composing features. Our work exposed a fundamental relationship between model derivation and model refinement in FOMDD, which we expressed as a commuting diagram/relationship. We exploited this relationship — the transformation of a composition of models equals the composition of transformed models — to validate the correctness of our domain abstractions, tools, and portlet specifications at a cost of longer synthesis times. The relationship could also be used to reduce synthesis times if validation is not an issue.

While the portlet domain admittedly has specific and unusual requirements, there is nothing domain-specific about the need for MDD and FOP and their benefits. In this regard, PMDD is not unusual; it is an example of many domains where both technologies naturally complement each other to produce a result that is better than either could deliver in isolation. FOMDD offers a fresh perspective in program and product-line synthesis where mathematical properties — in addition to engineering feats — guide a principled design of complex systems. Research on MDD and FOP should focus on infrastructures that support their integration, and researchers should be cognizant that their synergy is not only possible, but desirable.

Acknowledgments. We thank F. I. Anfurrutia, M. Azanza, D. Benavides, and J. Gray for their helpful comments on earlier drafts of this paper. This work was co-supported by the Spanish Ministry of Science & Education, the European Social Fund under contract TIC2005-05610 and by NSF's Science of Design Project #CCF-0438786. Trujillo has a doctoral grant from the Spanish Ministry of Science & Education. This work was partially done while Trujillo was visiting the University of Texas at Austin.

8 References

[1] M. Anastaspoulos, et. al. "Optimizing Model-Driven Development by Deriving Code Generation Patterns from Product Line Architectures". *NetObject Days 2005*.

[2] Apache `ant`. `http://ant.apache.org/`

[3] L. Baresi, F. Garzotto and P. Paolini. "From Web Sites to Web Applications: New Issues for Conceptual Modeling". *ER Workshops 2000*.

[4] D. Batory, G. Chen, E. Robertson, and T. Wang. "Design Wizards and Visual Programming Environments for GenVoca Generators". *IEEE TSE*, May 2000.

[5] D. Batory, et al. "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study". *ACM TOSEM,* April 2002.

[6] D. Batory, J.N. Sarvela, and A. Rauschmayer. "Scaling Step-Wise Refinement". *IEEE TSE*, June 2004.

[7] D. Batory. "Feature Models, Grammars, and Propositional Formulas". *SPLC 2005*.

[8] D. Batory. "Multi-Level Models in Model Driven Development, Product-Lines, and Metaprogramming". *IBM Systems Journal*, Volume 45, Number 3, 2006.

[9] D. Batory. "From Implementation to Theory in Program Synthesis". Keynote at *POPL 2007*.

[10] J. Bezivin. "Model Driven Engineering: Principles, Scope, Deployment, and Applicability". *GTTSE 2005*.

[11] G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, and B. Selic. "The IBM MDA Manifesto". *The MDA Journal*, May 2004.

[12] S. Ceri, P. Fraternali and M. Matera. "Conceptual Modeling of Data-Intensive Web Applications". *IEEE Internet Computing*, vol. 6, no. 4, pp. 20-30. 2002.

[13] K. Czarnecki, M. Antkiewicz, et al. "Model-Driven Software Product-Lines". *OOPLSA 2005 Posters*.

[14] K. Czarnecki and M. Antkiewicz. "Mapping Features to Models: A Template Approach Based on Superimposed Variants". *GPCE 2005*.

[15] S. Deelstra, M. Sinnema, J. van Gurp, and J. Bosch. "Model Driven Architecture as Approach to Manage Variability in Software Product Families". *MDAFA 2003 Workshop*.

[16] O. Diaz and J.J. Rodriguez. "Portlets as Web Components: an Introduction". *Journal of Universal Computer Science*, 10(4):454-472. April 2004.

[17] O. Diaz, S. Trujillo and S. Perez. "Turning Portlets into Services: Introducing the Organization Profile". *WWW 2007*.

[18] Exo Portal Platform. `http://www.exoplatform.com/`

[19] M. C. Ferreira de Oliveira, M. A. Santos Turine, and P. C. Masiero. "A Statechart-Based Model for Hypermedia Applications". *ACM TOIS*, January 2001.

[20] B. González-Baixauli, M.A. Laguna, and Y. Crespo. "Product Lines, Features, and MDD". *EWMT 2005 Workshop*.

[21] J. Gray. et al. "Model Driven Program Transformation of a Large Avionics Framework". *GPCE 2004*.

[22] D. Harel. "Statecharts: A visual formalism for complex systems". *Science of Computer Programming*, 8(3), 1987.

[23] Java Community Process. *JSR 168 Portlet Specification*, October 2003. http://www.jcp.org/en/jsr/detail?id=168.

[24] A. Kleppe, J. Warmer, and W. Bast. "*MDA Explained: The Model Driven Architecture: Practice and Promise*", Addison-Wesley, 2003.

[25] N. Koch and A. Kraus. "The Expressive Power of UML-based Web Engineering". *IWWOST 2002*.

[26] I. Kurtev and K. van den Berg. "Building Adaptable and Reusable XML Applications with Model Transformations". *WWW 2005*.

[27] G. Leavens, et al. "Roadmap for Enhanced Languages and Methods to Aid Verification", ftp://ftp.cs.iastate.edu/pub/techreports/TR06-21/TR.pdf

[28] J. Lee, N.L. Xue, and T.L. Kuei. "A Note on State Modeling through Inheritance", *SIGSOFT Soft. Eng. Notes 23 (1998)*.

[29] A.T. McNeile and N. Simons. "State Machines as Mixins". *Journal of Object Technology 2, 2003*.

[30] T. Mens, K. Czarnecki, and P. van Gorp. "A Taxonomy of Model Transformations", Dagstuhl Seminar Proceedings 04101 http://drops.dagstuhl.de/opus/volltexte/2005/11.

[31] A. Narayanan and G. Karsai. "Towards Verifying Model Transformations". In *GT-VMT workshop at ETAPS 2006*.

[32] OASIS. *Web Service for Remote Portals (WSRP)* Version 1.0. 2003. http://www.oasis-open.org/committees/wsrp/

[33] OMG. *Unified Modeling Language (UML)*, version 2.0. 2005. http://www.uml.org/#UML2.0

[34] OMG. *XML Metadata Interchange Mapping Specification*, version 2.1, 2005, http://www.omg.org/technology/documents/formal/xmi.htm

[35] O. Pastor et al. "The OO-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming". *Inf. Syst*., vol. 26, no. 7, 2001.

[36] F. Paterno and C. Mancini. "Model-Based Design of Interactive Applications". *ACM Intelligence*, pp. 27-37. 2000.

[37] B. Pierce. *Basic Category Theory for Computer Scientists*, MIT Press, 1991.

[38] D. Schmidt, A. Nechypurenko, and E. Wuchner. "MDD for Software Product-Lines: Fact or Fiction", *Models 2005 Workshop 9*.

[39] P. Selinger, et al. "Access Path Selection in a Relational Database System", *ACM SIGMOD*, 1979.

[40] N. Serrano and I. Ciordia. "Ant: Automating the Process of Building Applications". *IEEE Software*, 21(6):89-91, November/December 2004.

[41] S. Trujillo, D. Batory, and O. Diaz. "Feature Refactoring a Multi-Representation Program into a Product Line". *GPCE 2006.*

[42] S. Trujillo. Feature Oriented Model Driven Product Lines. Ph.D. thesis. University of the Basque Country. 2007.

[43] W3C. State Chart XML (SCXML): State Machine Notation for Control Abstraction, *W3C Working Draft* 24 January 2006. http://www.w3.org/TR/scxml/.

[44] W3C. *Simple Object Access Protocol (SOAP)* 1.1, June 2003. http://www.w3.org/TR/soap12/.

[45] W. Weber and P. Metz. "Reuse of Models and Diagrams of the UML and Implementation Concepts Regarding Dynamic Modeling". In *The Unified Modeling Language: Technical Aspects and Applications*, 190-203, 1998. Physica-Verlag.

[46] MyYahoo. http://my.yahoo.com/.

9 Appendix: Transforming Refinements

Let feature function F be defined by the tuple: $\langle \Delta F_{sc}, \Delta F_{act\text{-}usr}, \Delta F_{view\text{-}usr}, \Delta F_{jak\text{-}usr}, \Delta F_{jsp\text{-}usr} \rangle$. We can map F to a ΔF_{raw} in the following way. First, we define a new transformation ($T'_{sc2ctrl}$) that maps a refinement of a state chart (ΔF_{sc}) to a refinement or *delta* of a PSL controller (ΔF_{ctrl}):

$$\Delta F_{ctrl} = T'_{sc2ctrl}\ (\Delta F_{sc}) \quad (21)$$

Second, we need other transformations to map ΔF_{ctrl} to an action skeleton delta ($\Delta F_{act\text{-}sk}$) and a view skeleton delta ($\Delta F_{view\text{-}sk}$):

$$\Delta F_{act\text{-}sk} = T'_{ctrl2act}\ (\Delta F_{ctrl}) \quad (22)$$
$$\Delta F_{view\text{-}sk} = T'_{ctrl2view}\ (\Delta F_{ctrl}) \quad (23)$$

Third, we composed the action skeleton delta ($\Delta F_{act\text{-}sk}$) computed above with a hand-written refinement ($\Delta F_{act\text{-}usr}$) to yield a complete PSL action delta (ΔF_{act}). The same applies to producing a complete PSL view delta (ΔF_{view}) by composing its skeleton and hand-written refinement:

$$\Delta F_{act} = \Delta F_{act\text{-}usr} \bullet \Delta F_{act\text{-}sk} \quad (24)$$
$$\Delta F_{view} = \Delta F_{view\text{-}usr} \bullet \Delta F_{view\text{-}sk} \quad (25)$$

Given these deltas (ΔF_{act}, ΔF_{view}), we wrote additional transformations to map them to their delta Jak and Jsp code skeleton counterparts:

$$\Delta F_{jak\text{-}sk} = T'_{act2jak}\ (\Delta F_{act}) \quad (26)$$
$$\Delta F_{jsp\text{-}sk} = T'_{view2jsp}\ (\Delta F_{view}) \quad (27)$$

and composed them with their code refinements:

$$\Delta F_{jakcode} = \Delta F_{jak\text{-}usr} \bullet \Delta F_{jak\text{-}sk} \quad (28)$$
$$\Delta F_{jspcode} = \Delta F_{jsp\text{-}usr} \bullet \Delta F_{jsp\text{-}sk} \quad (29)$$

The delta raw material that a feature F adds to its base is:

$$\Delta F_{raw} = \{\Delta F_{ctrl}, \Delta F_{act}, \Delta F_{view}, \Delta F_{jakcode}, \Delta F_{jspcode}\} \quad (30)$$

where the components of ΔF_{raw} are derived by (21)-(30).