International Confernce on Model Transformations (ICMT) 2008

Lifting Transformational Models of Product Lines: A Case Study

Greg Freeman[1], Don Batory[2], and Greg Lavender[2]

[1]Dept. of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas 78712 U.S.A.
gfreeman@ece.utexas.edu

[2]Dept. of Computer Sciences
University of Texas at Austin
Austin, Texas 78712 U.S.A.
{batory,lavender}@cs.utexas.edu

Abstract. *Model driven development (MDD)* of *software product lines (SPLs)* merges two increasing important paradigms that synthesize programs by transformation. MDD creates programs by transforming models, and SPLs elaborate programs by applying transformations called features. In this paper, we present the design and implementation of a transformational model of a product line of scalar vector graphics and JavaScript applications. We explain how we simplified our implementation by lifting selected features and their compositions from our original product line (whose implementations were complex) to features and their compositions of another product line (whose specifications were simple). We used operators to map higher-level features and their compositions to their lower-level counterparts. Doing so exposed commuting relationships among feature compositions in both product lines that helped validate our model and implementation.

Keywords. transformation reuse, code generation, model composition, hgh-level transformations, features, product-lines.

1 Introduction

Model driven development (MDD) offers the potential to automate manual, error prone, and time intensive tasks and replace them with high level modeling and code generation. Modeling software has a number of advantages including strategically approaching problems top-down, documenting software structure and behavior, and reducing the time and cost of application development. *Feature oriented programming (FOP)* solves a complementary problem of building families of similar programs (a.k.a. *software product lines (SPL)*). Features are increments in program development and are transformations (i.e., functions that map a program to a more elaborate program). Both paradigms naturally invite simple descriptive models of program construction that are pure-

ly transformation-based (i.e., program designs are expressed as a composition of functions) and their integration is synergistic [32].

Our paper makes two contributions. First, we explain how we designed and implemented a product line of *scalar vector graphics (SVG)* and JavaScript applications. Our approach combines FOP and MDD in a way that allows us to use the language of elementary mathematics to express our approach in a straightforward and structured way, and to illustrate how transformational models of SPLs can be defined and implemented. Second, we explain how we simplified our effort by lifting selected features and their compositions from our original product line (whose implementations were complex and tedious) to features and their compositions to another product line (whose specifications were simple). Mathematical expressions define transformation paths that combine feature composition and model translation, exposing commuting relationships among transformations that helped validate our model and implementation. We begin with an overview of the domain of our case study.

2 MapStats

MapStats is an application that displays population statistics for different US states using SVG and JavaScript [26]. *Scalar vector graphics (SVG)* is a *World Wide Web Consortium (W3C)* language for describing two dimensional graphics and graphical applications. JavaScript is a scripting language that can be embedded within SVG to generate dynamic content.

MapStats displays an interactive map of the US, as shown in Fig. 1. Users can alter the map to selectively display rivers, lakes, relief, and population diagrams. A map navigator allows users to zoom and pan the primary map.

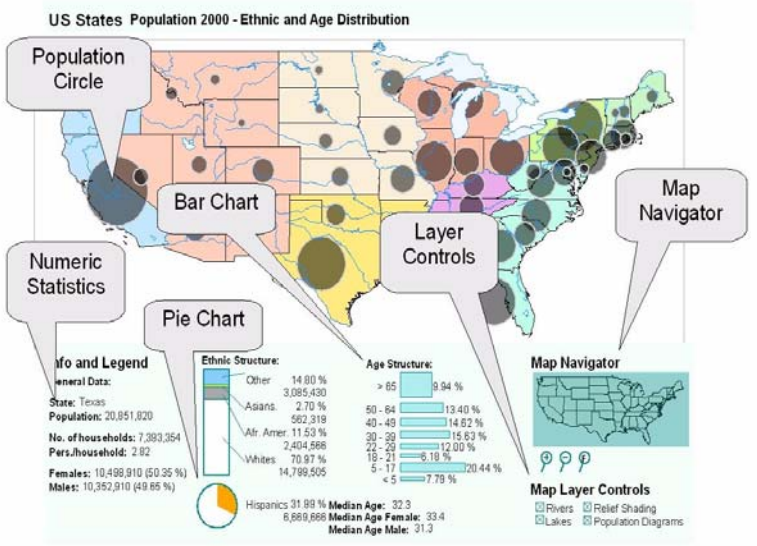


Fig. 1. MapStats SVG Case Study Application with all Features

When a user mouses over a state, various population statistics for the state are shown in text and graphical charts. Demographic attributes can be based on sex, age, and race. Statistics with charts can also be shown.

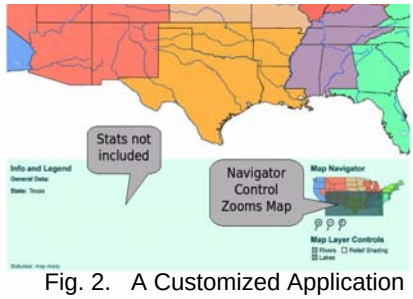
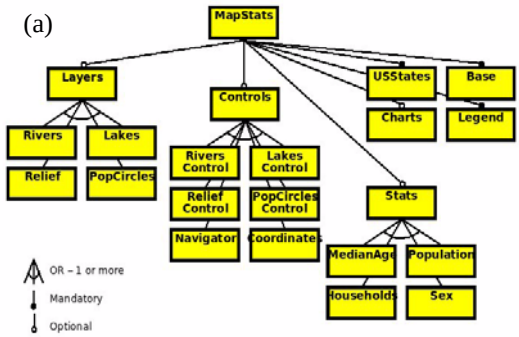


Fig. 2. A Customized Application

We refactored MapStats into a base application and optional features to allow a product line of variants to be created by composing the base with desired features. Fig. 2 shows a customized MapStats application that excludes statistical charts.

Feature diagrams are a standard way to express a product line [12][19]. A *feature diagram* is an and-or tree, where terminals represent primitive features and non-terminals are compound features. Fig. 3a shows a portion of the feature diagram for the `MapStats` product line; Fig. 3b lists the actual names and descriptions of the features that we created. (Not shown in Fig. 3 are the compatibility constraints among features, i.e., selecting one feature may require the selection or deselection other features [5][12]). `MapStats` features include: each statistic that can be displayed, each map layer, each map control, and run-time display options. For example, the `Rivers` feature adds rivers to the map of US states and the `RiversControl` feature adds a control that lets the user turn the river layer on and off at run time.

(a)



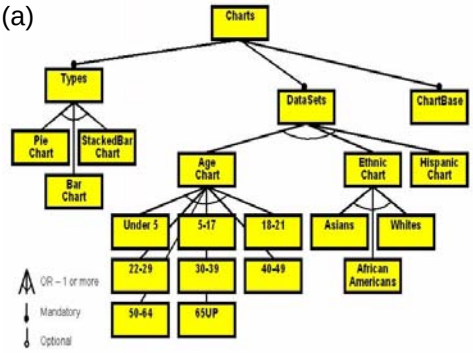
(b)

Feature	Description
Base	The base application
USStates	Displays map of US States
Legend	Adds chart displays and statistics
Charts	Adds charts
Households	Displays the number of households/state
Sex	Displays the ratio of males to females
MedianAge	Displays the median age
Population	Displays the total population
Navigator	Adds a control to let user pan and zoom the map
Coordinate	Shows the xy coordinates of the mouse
Relief	Adds relief to the map
PopCircles	Adds population circles to indicate the population of each state
Rivers	Adds rivers to the map
Lakes	Adds lakes to the map
ReliefControls	Adds a control to turn relief on and off
PopCirclesControls	Adds a control to turn population circles on and off
RiversControls	Adds a control to turn rivers on and off
LakesControls	Adds a control to turn lakes on and off

Fig. 3. MapStats Feature Diagram and Feature Descriptions

Again, Fig. 3a is a portion of the feature diagram for MapStats. We further decomposed the terminal `Charts` feature of Fig. 3a into a product line of charts. Fig. 4a shows its feature diagram and Fig. 4b lists the actual names and descriptions of the `Charts` features that we created. `Charts` features used three data sets: age, ethnic, and Hispanic. (The Hispanic data set was an artifact of the original application which we left intact). We used features to specify chart types: bar, stacked-bar, and pie. The combination of

(a)



(b)

Feature	Description
ChartBase	An empty collection of charts
Pie	Creates a pie chart for each data set
Bar	Creates a bar chart for each data set
StackedBar	Creates a stacked-bar chart for each data set
Age	Creates charts with age data for each chart type, grouped by age ranges
Under5	Adds under 5 age group
5-17	Adds 5-17 age group
18-21	Adds 18-21 age group
22-29	Adds 22-29 age group
30-39	Adds 30-39 age group
40-49	Adds 40-49 age group
50-64	Adds 50-64 age group
65UP	Adds 65 and up age group
Ethnic	Creates charts with ethnic data
Hispanic	Adds Hispanic data
Asians	Adds Asians data
AfricanAmerican	Adds African American data
Whites	Adds Whites data

Fig. 4. Chart Feature Model and Feature Descriptions

chart types and data sets specified whole charts. So if two data sets and two chart types were specified, four charts would be created representing each combination.

Thus, we began our design in the standard way: we created a feature diagram for our product line. The next step was to implement features as transformations.

3 A Transformation-Based Model of Product Lines

GenVoca is a compositional paradigm and methodology for defining product lines solely by transformations: *it does not promote any particular implementation technology or tool*. Instead, it stresses that adding a feature to a program (however the program is represented) is a transformation that maps the original program to an extended program. There is a long history of creating and implementing GenVoca product lines in different domains (e.g. [7][8]). We review its key ideas and then explain our model of MapStats.

3.1 GenVoca

A GenVoca representation is a set of base programs and features (transformations) that extend or elaborate programs. The GenVoca representation expresses which features are used to compose a product line instance and the valid combinations of features in a product line. An example model `G={f,h,i,j}` contains the following parts: Base programs are values (0-ary functions):

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

and unary functions (transformations) are features:

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

• denotes function composition. The design of a program is expression:

```
p1 = j•f      // program p1 has features j and f
p2 = j•h      // program p2 has features j and h
p3 = i•j•h    // program p3 has features i, j, and h
```

The set of programs defined by a GenVoca model is its *product line*. Expression optimization is program design optimization, and expression evaluation is program synthesis [6][29]. Tools that validate feature compositions are discussed in [5][30]. Note that features (transformations) are reusable: a feature can be used in the creation of many programs in a product line.

A fundamental characteristic of features is that they "cross-cut" implementations of base programs and other features. That is, when a feature is added to a program, new classes can be added, new members can be added to existing classes, and existing methods can be modified. There is a host of technologies — including aspects, languages for object-oriented collaborations, and rewrite rules in program transformation systems — that can modularize and implement features as transformations. In MapStats, features not only refine JavaScript programs by adding new classes, methods and statements, but also new graphics elements can be added to SVG programs.

The relationship of a GenVoca model (i.e., 0-ary and unary functions) to a feature diagram is straightforward: each terminal of a feature diagram represents either a base program or a unary function. Compound features correspond to GenVoca expressions.

3.2 A Model of MapStats

A GenVoca model of MapStats has a single value (`Base` of Fig. 3); its unary functions are the remaining features of Fig. 3 and the features of the Charts feature diagram:

```
MapStats = { Base, USStates, ...    // features from Fig. 3
             ChartBase, Pie, ... }  // features from Fig. 4
```

To simplify subsequent discussions, instead of using the actual names of `MapStats` features, let us use subscripted letters. M_0 is the base program of `MapStats`, $M_1..M_n$ are the (unary function) features of the MapStats feature diagram and $C_0...C_m$ are (unary function) chart features:

```
MapStats = { M0 ... Mn,             // features from Fig. 3
             C0 ... Cm }            // features from Fig. 4
```

An application `A` in the `MapStats` product line is an expression:

$$A = (C_2 \bullet C_1 \bullet C_0) \bullet M_1 \bullet M_0 \quad (1)$$

That is, application `A` is constructed by elaborating base program M_0 with a sequence of `M` features followed by a sequence of `C` features, where subexpression $(C_2 \bullet C_1 \bullet C_0)$ synthesizes the JavaScript that displays one or more charts. The original MapStats application `Orig`, which is part of our product line, is synthesized by composing all features:

$$Orig = (C_m \bullet ... \bullet C_0) \bullet M_n \bullet ... \bullet M_0$$

Each `MapStats` feature can encapsulate SVG and JavaScript refinements (crosscuts) of the base application (M_0).

3.3 Implementation Overview

Our implementation of `MapStats` was straightforward. Our base program (M_0) was a pair of SVG and JavaScript programs. Each `MapStats` feature (M_i) could modify the SVG program, the JavaScript program, or both. We used the *AHEAD Tool Suite (ATS)*

to implement `MapStats` features [1], and in particular, the XAK tool as the composition operator.

XAK is a language to refine XML documents and is also a tool to compose XML documents with their refinements `[3]`. A XAK base document is an XML file containing labeled *variation points* or *join points* to identify positions in a document where modifications can take place. A XAK refinement (unary function) is an XML file that begins with a `refine` element. Its children define a set of modifications, where each modification pairs an XPath expression with an XML fragment. The XPath expression identifies variation points or join points in an XML document, and the XML fragment is appended as a child node of the selected parent node(s). XAK can also prepend, replace, and delete nodes as well as perform operations on attributes, sibling nodes, and text nodes, however, our need was limited to child node appending.

To illustrate, Fig. 5a shows an elementary base document; Fig. 5b is a XAK refinement that appends an XML tree as another child of `<mynode>`. In *Aspect Oriented Programming (AOP)* terms, '`xr:at`' node specifies a pointcut as an XPath expression, which in this case looks for nodes called '`mynode`'. The '`xr:append`' node defines the advice action and body. The action for this example is to append '`mychildnode`' with a data attribute of '`2`'. Applying the refinement to the base yields the composite document of Fig. 5c.[1]

```
<mynode>
  <mychildnode data="1">
  </mychildnode>
</mynode>
```

(a) base

```
<xr:refine xmlns:xr="http://
www.atarix.org/xmlRef ...
  <xr:at select="//mynode">
    <xr:append>
      <mychildnode  data="2">
      </mychildnode>
    </xr:append>
  </xr:at>
</xr:refine>
```

(b) refinement

```
<mynode>
  <mychildnode data="1">
  </mychildnode>
  <mychildnode data="2">
  </mychildnode>
</mynode>
```

(c) composed

Fig. 5. XAK base, refinement, and Composition

As SVG documents are XML documents, XAK provided the language and tool for SVG document modification. However, ATS does not have a language to express JavaScript refinements, and a tool to compose refinements with a base JavaScript program. To circumvent this, we used XML to encode both JavaScript and JavaScript refinements, and used XAK to compose them. The resulting JavaScript program was produced by stripping XML tags.

4 Lifting

It quickly became evident that `MapStat` chart features $C_0 \ldots C_m$ were extremely tedious to write. We applied a key principle of MDD to save us effort: we created a high level

1. Aspects can be implemented by transformations; aspect compilers transform an input program to a “woven” program where additional code has been appropriately inserted [23].

DSL to specify charts and their features. Fig. 6 shows a fragment of a chart spec. A chart XML element defines a chart and an item defines an element in the chart. XML attributes can change the type of chart (pie, bar, or stacked-bar) as well as the names, colors, and field attribute codes for chart items.

```
<chart data-type="age-population" type="pieChart" ...
  <item attr="AGE_30_39" color="lightgreen" name= ...
  <item attr="AGE_22_29" color="lightcyan" name=...
</chart>
```

Fig. 6. A Chart Spec Fragment

Given chart specs, it is easy to write chart features (transformations). For example, a XAK refinement of Fig. 6 that appends the age data item for 18-21 is shown in Fig. 7. The underlined node defines a pointcut (XPath expression) that identifies all charts with the attribute `@datatype='age-population'`; such a chart would have the item `AGE_18_21` appended to it. (In AOP-speak, this advice is homogenous [11]).

```
<xr:refine xmlns:xr="http://www.ataric.org/xmlRef ...
  <xr:at select="//chart[@data-type='age-population' ...
    <xr:append>
      <item attr="AGE_18_21" color="cyan" ...
    </xr:append>
  </xr:at>
</xr:refine>
```

Fig. 7. Example Chart Feature

We wrote XSLT transformations to map a chart spec (or chart spec refinement) to its corresponding `MapStat` chart feature implementation (i.e., a JavaScript refinement). XSLT was chosen for the translation step since our models were XML-based. The image that is represented by the composite chart (Fig. 6 composed with Fig. 7) is shown in Fig. 8 where all three age groups are displayed. In general, we found a chart DSL specifications to be 4-10 times shorter than their generated JavaScript counterparts.

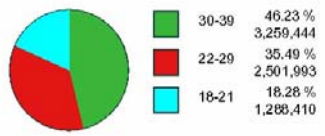


Fig. 8. Pie Chart with Three Age Categories

By *lifting* (raising) the level of abstraction of chart feature implementations, in effect what we did was create another product line — a product line of charts. That is, we lifted the chart features $C_0 \ldots C_m$ of `MapStats` into a separate GenVoca model called `Charts`:

$$\texttt{Charts} = \{ S_0 \ \ldots \ S_m \}$$

where S_0 was the base chart spec, and each `Charts` feature S_i was a chart spec refinement. `Charts` features are in 1-to-1 correspondence with their `MapStats` chart features. XSLT transformations τ and τ' defined this correspondence:

$$C_0 = \tau(S_0) \qquad (2)$$

$$C_i = \tau'(S_i) \quad \text{// for all } i=1..m \qquad (3)$$

τ and τ' have very similar implementations: their difference is due to the type of their argument: τ maps a `Charts` *value* to a `MapStats` function (i.e., JavaScript refinement); τ' maps a `Charts` *function* to a `MapStats` function.

Note that an *operator* maps an input function to an output function. τ' is an operator that maps a `Charts` refinement transformation to a `MapStats` refinement transformation. τ maps a `Charts` 0-ary function S_0 to the `MapStats` unary function C_0. Operators τ and τ' have a basic commuting relationship which we explain in Section 6.

Even though we now used lifted features, the way we specified a target `MapStats` application changed minimally. We still used the original feature diagram of MapStats to specify a `MapStats` application and to create its GenVoca expression (which starts with the base program M_0 and applying `MapStats` features to elaborate it). But instead of implementing chart features $C_0 \ldots C_m$ directly in terms of JavaScript refinements, we used chart specs and chart refinements $S_0 \ldots S_m$. To synthesize a `MapStats` application `A` (equation `(1)`), we rewrote its expression using `(2)` and `(3)`:

$$\begin{aligned} A &= (C_2 \bullet C_1 \bullet C_0) \bullet M_1 \bullet M_0 && \text{// original MapStats expr} \\ &= \tau'(S_2) \bullet \tau'(S_1) \bullet \tau(S_0) \bullet M_1 \bullet M_0 && \text{// rewrite} \qquad (4) \end{aligned}$$

and evaluated `(4)` to synthesize `A`. We call the raising of features and their compositions from one product line to another *lifting*. Lifting can be applied to any GenVoca product line. Transformations (like τ and τ') are used to define maps between unlifted features and their lifted counterparts. Constraints that govern the composition of original `MapStats` features remain unchanged.

5 Implementation Details

In this section, we illustrate some of the features and mappings discussed earlier, in order to make our discussions concrete.

A chart spec defines one or more charts. Each chart is implemented by a unique JavaScript class. For example, a pie chart that displays age information that includes the range of 18-21 is defined as a JavaScript class (below named `agePie`) that has a method (`buildData`) that populates this particular data set:

```
function agePie() {                 // JavaScript class definition
  ...
  this.buildData = function() {     // buildData method
    ...
    this.chartAttrArray.push("AGE_18_21");
    this.chartNameArray.push("18-21");
    this.chartColorArray.push("cyan");
    ...
  }
}
```

At run-time, a JavaScript object is created for each chart, populated with data, and then displayed:

```
var agepie = new agePie();          // instantiate object
agepie.buildData();                 // populate data
agepie.showData();                  // display
```

To see how this JavaScript class was synthesized, let's look at a `Charts` feature expression that could generate it:

```
AGE_18_21•Age•Pie•ChartBase
```

That is, the chart spec begins with `ChartBase`, it is refined to a pie chart that displays age information (`Age•Pie`), and then the age category 18-21 is added. Internally, our tools generate unique names for each chart. The manufactured name given to the chart of our example is "`agePie`".

Let's now focus on the `AGE_18_21` feature. The XAK refinement that defines it was depicted in Fig. 7, which we reproduce below:

```
<xr:refine xmlns:xr="http://www.atarix.org/xmlRef ...
  <xr:at select="//chart[@data-type='age-population' ...
    <xr:append>
      <item attr="AGE_18_21" color="cyan" ...
    </xr:append>
  </xr:at>
</xr:refine>
```

This transformation adds the age category 18-21 to all charts of a charts spec that display age information. In our example, there is only one chart, `agePie`. Note that the underlined code denotes the pointcut (XPath expression) that captures the relevant charts to modify.

Let's see the result of transforming the `AGE_18_21` `Charts` feature into its corresponding `MapStats` feature (denoted $\texttt{AGE\_18\_21}_{\texttt{mapstats}}$). The τ' operator maps `AGE_18_21` to $\texttt{AGE\_18\_21}_{\texttt{mapstats}}$, where a fragment of $\texttt{AGE\_18\_21}_{\texttt{mapstats}}$ is:

```
<xr:refine ... >
  <xr:at select="//function[@data-type='age-population']
       [@parentId='ChartArea2'][@name='buildData']"...>
    <xr:append>
       <statement>
    this.chartAttrArray.push("AGE_18_21");
    this.chartNameArray.push("18-21");
    this.chartColorArray.push("cyan");
       </statement>
    </xr:append>
  </xr:at>
</xr:refine>                                                    (5)
```

That is, the above XAK refinement adds the JavaScript code in *`italic red`* to the `buildData` method of each JavaScript class of a chart that displays age information. Note that the underlined code denotes the pointcut (XPath expression) that captures the relevant `buildData` methods. So the translation of `AGE_18_21` to $\texttt{AGE\_18\_21}_{\texttt{mapstats}}$ maps a pointcut (XPath expression) whose joinpoints are in chart specs to a pointcut whose joinpoints are in JavaScript programs. Also, the addition of a chart element is mapped to the addition of statements in the JavaScript method `buildData`.

As mentioned earlier, operators τ and τ' are implemented in XSLT. They look for patterns in charts specifications and instantiate JavaScript code templates. For example, when a '`chart`' element is encountered in a chart spec, a corresponding JavaScript class

is added with the methods `buildData` and `showData`. When an '`item`' element is found in a chart spec, statements are added to an appropriate JavaScript method. As an example, a fragment of the XSLT definition of τ' is shown below:

```
<xsl:template match="xr:at/xr:append/c:item">
     ... map Charts pointcut to MapStats pointcut...
     <xr:at select="{$path}">
      <xr:append>
        <xsl:variable name="attr" select="@attr"/>
        <xsl:variable name="color" select="@color"/>
        <xsl:variable name="name" select="@name"/>
        <statement>
        this.chartAttrArray.push("<xsl:value-of select="$attr"/>");
        this.chartNameArray.push("<xsl:value-of select="$name"/>");
        this.chartColorArray.push("<xsl:value-of select="$color"/>");
        </statement>
      </xr:append>
    </xr:at>
</xsl:template>                                                    (6)
```

Note that the code in *italic red* is a template whose parameters are provided by the input to τ'. In our example, the `AGE_18_21` input to τ'assigns the value `AGE_18_21` to `attr,18-21` to `name`, and `cyan` to `color`. The *italic red* code of `(5)` is generated by instantiating the τ' template with these parameters. By writing a general transformation τ' once and reusing it (to translate other `Charts` features that were differentiated only by their parameters), saved us considerable effort as mentioned earlier. Notice also that part of τ' is to map the pointcut of a charts spec to a corresponding pointcut that captures the corresponding JavaScript methods. This mapping is done via string manipulation, which we elide the details, and indicate by underlined code in `(6)`.

6 Commuting Relationships

Lifting defines a commuting relationship between `Charts` features and `MapStats` features that relate τ and τ' and that offers us yet another way to synthesize `MapStats` applications. Instead of separately translating each `Charts` feature S_i to its C_i counterpart as we did in `(4)`, we could synthesize a composite chart spec `S` (e.g., $S=S_2 \bullet S_1 \bullet S_0$) by starting with a base spec S_0, and add features S_1 and S_2, and *then* transform `S` into its corresponding JavaScript implementation. That is, another way to synthesize application `A` is:

$$A = \tau(S_2 \bullet S_1 \bullet S_0) \bullet M_1 \bullet M_0 \qquad (7)$$

The equivalence of `(4)` and `(7)` is due to the commuting relationship:

$$\tau(S_i \bullet S) = \tau'(S_i) \bullet \tau(S) \qquad (8)$$

where `S` is a `Charts` expression and S_i is a `Charts` feature. (8) says composing `Charts` features and translating to a `MapStats` representation equals translating each `Chart` feature separately and composing. *The value of commuting relationships is that they define properties of valid implementations of transformational models of product lines. The correctness of a model and tools is demonstrated when its commuting relationships are demonstrated. Commuting relationships provide a simple means to ex-*

press and compare different methods of applying transformations and transformation of transformations (i.e., operators).

> **Note**: a general name for (8) is a homomorphism: given two sets X and Y and a single binary operation on them, a *homomorphism* is a map $\Phi: X \rightarrow Y$ such that:
>
> $$\Phi(u \otimes v) = \Phi(u) \oplus \Phi(v) \quad (9)$$
>
> where $\otimes$ is the operation on X and $\oplus$ is the operation on Y. In MapStats, X is the Charts model and Y is the MapStats model; $\otimes$ and $\oplus$ both are $\bullet$. Homomorphisms define how expressions in one algebra are translated to expressions in another, i.e., (8)defines how Charts expressions are mapped to MapStats expressions.
>
> **Note**: what is the justification for (8)? Experimentally we have observed that compositions of features and derivations commute: when they do not, we find bugs in our transformation or tool chains. The commuting of features and derivations is an axiom of *Feature-Oriented MDD (FOMDD)* [31][32], which our work on MapStats is an example case study.

As we do not have formal models of Charts and MapStats, we do not have a proof of (8) for all Charts and MapStats features. Instead, we tested the correctness of (8). We synthesized multiple applications in two different ways (i.e., (4) and (7)) and then visually compared and executed both programs since (4) and (7) did not produce syntactically equivalent code. Graphical SVG applications with multiple transformation outputs allowed side-by-side visual comparison of many test cases. Other tests were performed with randomly selected features to ensure that each properly transformed the appropriately selected features. Although more sophisticated and thorough testing was possible (e.g., [24]), manual comparisons were sufficient for our goals.

Commuting relationships not only define properties that can be used to prove or test model and implementation correctness, but sometime they have additional benefits. We have observed in other domains that program synthesis can be substantially more efficient using one synthesis path (e.g., (4) or (7)) than another. For example, exploiting commuting relationships led to a 2-fold speed-up in synthesizing portlets [32], and over a factor of 10 in synthesizing test programs using Alloy [22]. Although we did observe trade-offs in building MapStats applications, they were not particularly significant. The utility of commuting relationships in MapStats was restricted to model and transformation validation.

7 Related Work

FOP and MDD paradigms have their historic roots in Lisp, which promoted the idea that programs are values (or “programs as data”) and transformations are functions that map values to values.

Combining MDD and product line transformations is not new [2][4][13][17][18][28][31][32]. Trujillo et al. used XAK and AHEAD to build web portlets from state chart models [32]. Our work builds upon theirs and provides further evidence that transformation-based models of product lines (that represent both features and model translations as transformations) expresses a general approach to software development. Al-

so, our use of lifting illustrates how basic concepts in elementary mathematics (e.g., operators and homomorphisms) lie at the core of program-development-by-transformations. The use of elementary mathematics as a language to express our design allows us to make this connection directly.

Trujillo et al. also apply model transformations that aid in the building of FOMDD (Feature Oriented Model Driven Development) applications, which include multiple transformation steps and different paths to generate an application [31].

Avila-García and others used transformations to apply features to models [4]. Their work focused on transformations of transformations that composed features for families of UML diagrams. Our work instead focuses on transforming high level models into executable applications.

Gonzalez-Baixauli and others have proposed using MDD to help product line engineers determine application variation points, and to assess the feasibility of automating software product line development with MDD [17]. Work by Deelstra and others have also used MDD as a means of identifying variations points within a product line [15]. Both papers infer that a feature could use *Platform Independent Model (PIM)* to *Platform Specific Model (PSM)* transformations to implement features that specify different platforms and implementation technologies.

Czarnecki and Helsen combined features and MDD in a different way by surveying different types of transformation methods and analyzing the various features of these methods [14]. Other prior work defined a taxonomy of different types of transformations and classified them as endogenous and exogenous [25]. Feature composition is an endogenous transformation, which uses the same source and target model representations. The τ and τ' transformations are exogenous, which use different source and target model (XML schema) representations.

Czarnecki and Antkiewicz connect features and behavioral models using model templates [13]. Model elements are tagged with predicates that reference features; the elements appear in a model instance when selected features satisfy the predicate. This is an alternative approach to artifact development in product lines; our approach stresses the modularization of features and their connection to transformations.

Kurtev uses XML transformations to develop XML applications [21]. The design of web applications includes functionality, content, navigation and presentation components.

Many results in MDD have laid a foundation for model transformations [9][10][20]. Even though this case study covers a specific domain and does not use UML model representations, model representations serve the same purpose of abstracting representations at different levels of detail. The `Charts` model representation is a type of PIM and the SVG and JavaScript model representations are types of PSMs.

8 Conclusions

We presented a product line of SVG+JavaScript applications that was defined and implemented solely in terms of transformations. Features of a product line were imple-

mented as transformations, and programs were specified as compositions of transformations. When we discovered that certain features were tedious to implement, we applied a basic principle of MDD to "lift" low-level implementations to DSL specifications and wrote transformations (operators) to map DSL specs (and their refinements) back to their SVG+JavaScript counterparts, ultimately saving effort.

What makes lifting interesting is its product line setting: we lifted selected features and their compositions from our original product line (`MapStats`) to features and compositions of another product line (`Charts`). We defined how features (transformations) in one product line could be transformed into features (transformations) of another via operators (τ and τ'). Doing so exposed commuting relationships between compositions of functions (i.e., tool chains and features). Such commuting relationships define properties of transformational models of program development; proving or validating (via testing) that these properties hold helps demonstrate model correctness. Our case study illustrated these ideas.

A primary reason why we were able to recognize commuting relationships and explain how features of one product line were related to another is that we used the language of elementary mathematics to express transformation-based designs of programs. Doing so enabled us to express our ideas in a straightforward and structured way and at the same time compactly illustrate how transformational models of software product lines can be defined, implemented, and explained.

Acknowledgements. We thank Prof. Hartmut Ehrig (University of Berlin), and Salvador Trujillo (University of the Basque Country) for their helpful comments on an earlier draft. We also thank the anonymous referees for their helpful insights. We gratefully acknowledge the support of the National Science Foundation under Science of Design Grants #CCF-0438786 and #CCF-0724979 to accomplish this work.

9 References

[1] AHEAD Tool Suite, `www.cs.utexas.edu/users/schwartz/index.html`

[2] Anastaspoulos, M., et al.: Optimizing Model Driven Development by Deriving Code Generation Patterns from Product Line Architectures. NetObject Days (2005)

[3] Anfurrutia, F. I., Diaz O., Trujillo, S.: On the Refinement of XML. ICWE (2007)

[4] Avila-García, O., García A.E., Redbull, E.V.S.: Using Software Product Lines to Manage Model Families in Model-Driven Engineering. SAC (2007)

[5] Batory, D.: Feature Models, Grammars, and Propositional Formulas. SPLC (2005)

[6] Batory, D., Robertson, E., Chen, G., Wang, T.: Design Wizards and Visual Programming Environments for Genvoca Generators. IEEE TSE (2000)

[7] Batory, D., O´Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM TOSEM (1992)

[8] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step Wise Refinement. IEEE TSE (2004)

[9] Bezivin, J.: Model Driven Engineering: Principles, Scope, Deployment, and Applicability. GTTSE (2005)

[10] Booch, G., Brown, A., Iyengar, S., Rumbaugh, J., Selic, B.: The IBM MDA Manifesto. The MDA Journal, (2004)

[11] Colyer, A., Rashid, A., Blair, G.: On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, Computing Department, Lancaster University (2004)

[12] Czarnecki, K., Eisenecker, U.: Generative Programming Methods, Tools, and Applications. Addison-Wesley, Boston, MA (2000)

[13] Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. GPCE (2005)

[14] Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems Journal, Vol. 45#3 (2006)

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

[16] Ehrig, H., Ehrig, K., Ermel, C., Hermann, F.: Taentzer, G.. Information Preserving Bidirectional Model Transformations. FASE (2007)

[17] Gonzlez-Baixauli, B., Laguna, M.A., Crespo, Y.: "Product Lines, Features, and MDD". EWMT Workshop (2005)

[18] Gray, J., et al.: Model Driven Program Transformation of a Large Avionics Framework. GPCE (2004)

[19] Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90TR-21 (1990)

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

[21] Kurtev, I., van den Berg, K.: Building Adaptable and Reusable XML Applications with Model Transformations. WWW (2005)

[22] Khurshid, S., Uzuncaova, E., Garcia, D., Batory, D.: Testing Software Product Lines Using Incremental Test Generation. Submitted.

[23] Lopez-Herrejon, R., Batory, D., and Lengauer, C.: A Disciplined Approach to Aspect Composition. PEPM (2006)

[24] Memon, A.M., Pollack, M.E., Soffa, M.L.: Using a Goal-driven Approach to Generate Test Cases for GUIs. ICSE (1999)

[25] Mens, T., Czarnecki, K., van Gorp, P.: A Taxonomy of Model Transformations. GraMoT (2005)

[26] Neuman, A.: US Population 2000: Ethnic Structure and Age Distribution, `http://www.carto.net/papers/svg/samples`

[27] Sabetzadeh, M., Easterbrook, S. M.: Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretic Approach. ASE (2003)

[28] Schmidt, D., Nechypurenko, A., Wuchner, E.: MDD for Software Product Lines: Fact or Fiction. Models, Workshop 8 at MODELS (2005)

[29] Selinger, P., et al.: Access Path Selection in a Relational Database System. ACM SIGMOD (1979)

[30] Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe Composition of Product Lines. GPCE (2007)

[31] Trujillo, S., Azanza, W., Diaz, O.: Generative Metaprogramming .GPCE (2007)

[32] Trujillo, S., Batory, D., Diaz, O.: Feature Oriented Model Driven Development: A Case Study for Portlets. ICSE (2007)