accepted for publication in Software and Systems Modeling August 2009. The original publication is available at www.springerlink.com

Lifting Transformational Models of Product Lines: A Case Study

Greg Freeman, Don Batory, Greg Lavender, Jacob Neal Sarvela

University of Texas at Austin
Austin, Texas 78712 U.S.A.
gfreeman@ece.utexas.edu
{batory,lavender,sarvela}@cs.utexas.edu

Abstract. *Model driven engineering (MDE) of software product lines (SPLs)* merges two increasing important paradigms that synthesize programs by transformation. MDE 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, high-level transformations, features, product lines, model driven engineering.

1 Introduction

Model driven engineering (MDE) 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 (SPLs)*). Features are increments in program development and are transformations (i.e., functions that map a simpler program to a more elaborate program). Both paradigms naturally invite descriptive models of program construction that are purely transformation-based (i.e., program designs are expressed as a composition of functions) and their integration is synergistic [50][51].

Our paper makes three contributions. First, we explain how we designed and implemented a product line of *scalar vector graphics (SVG)* and JavaScript applications. We combine FOP and MDE in a way that allows us to use the language of elementary mathematics to express our product line designs 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. Third, we illustrate the generality of lifting by relating it to our experiences in building the AHEAD Tool Suite, which is a very different product line than our SVG+JavaScript SPL. We begin with an overview of the domain of our primary case study.

2 MapStats

MapStats is an application that displays population statistics for different US states using SVG and JavaScript [39]. *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 selectively to display rivers, lakes, relief, and population diagrams. A map navigator allows users to zoom and pan the primary map.

When a user moves a mouse 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. We refactored MapStats into a base application and optional features to create a product line of variants by composing the

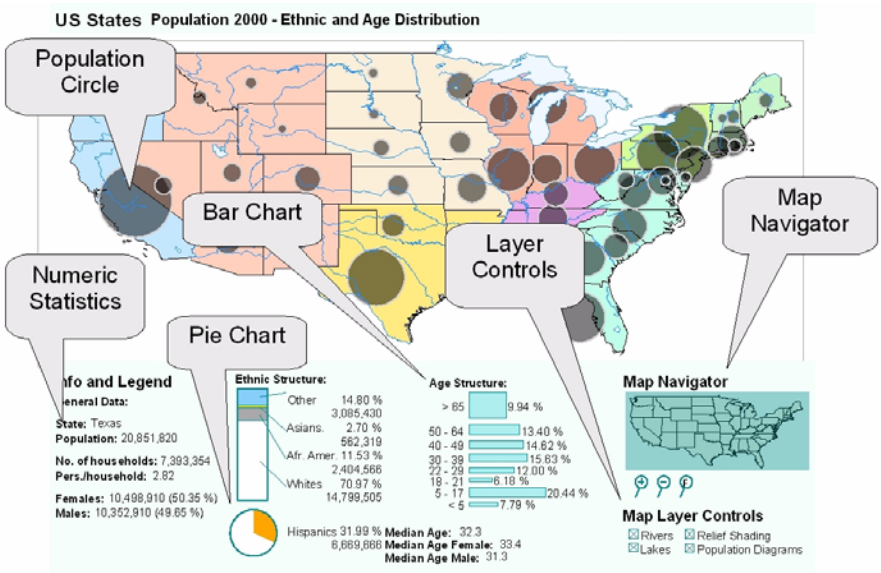


Fig. 1. MapStats Application With All Features

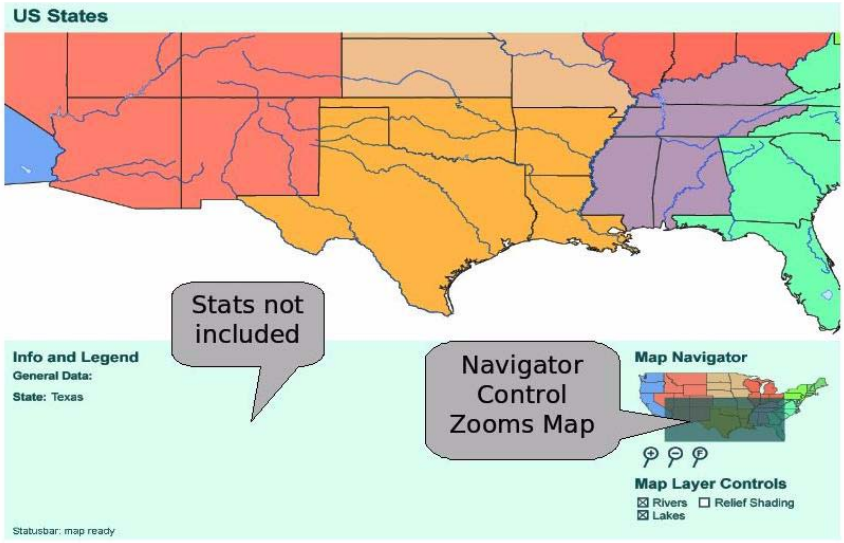


Fig. 2. A Customized MapStats Application

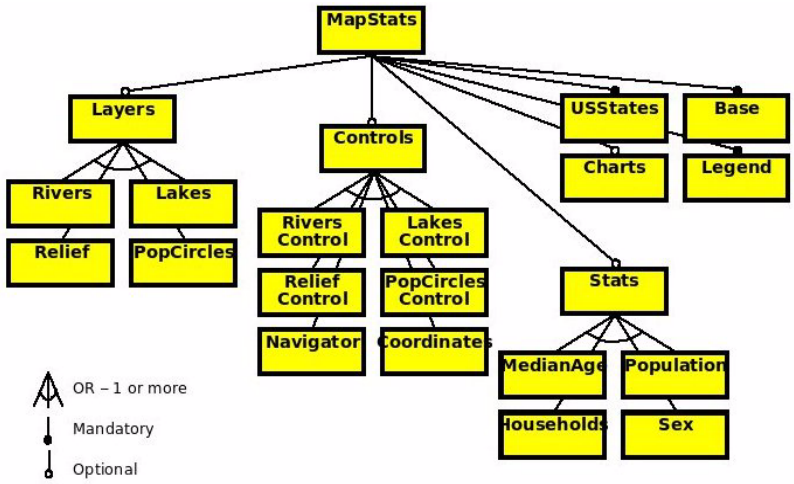
base with desired features. Fig. 2 shows a fragment of a customized MapStats application that excludes statistical charts.

Feature diagrams are a standard way to express a product line [16][29]. 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 of other features [6][16]). `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.

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 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.

(a)



(b)

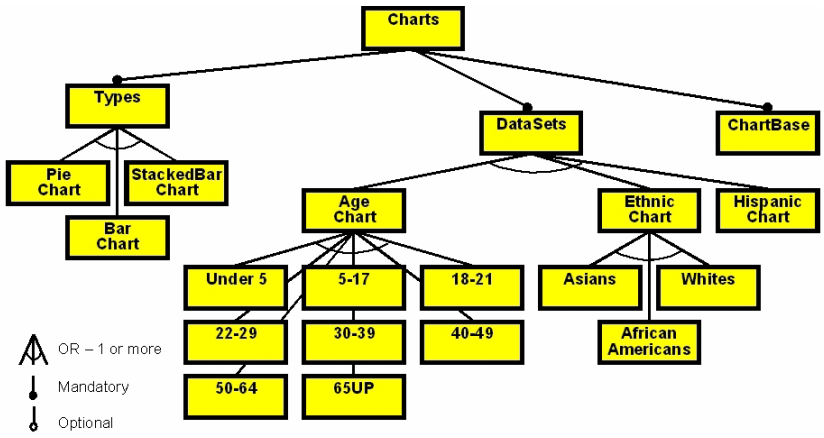
Feature	Description
Base	The base application
USStates	Displays map of US States
Legend	Adds chart displays and statistics
Charts	Adds charts
Households	Displays the number of households/state
Sex	Displays the ratio of males to females
MedianAge	Displays the median age
Population	Displays the total population
Navigator	Adds a control to let users pan and zoom the map
Coordinates	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
ReliefControl	Adds a control to turn relief on and off
PopCirclesControl	Adds a control to turn population circles on and off
RiversControl	Adds a control to turn rivers on and off
LakesControl	Adds a control to turn lakes on and off

Fig. 3. MapStats Feature Diagram and Feature Descriptions

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. [8][10]). We review its key ideas and then explain our model of MapStats.

(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

3.1 GenVoca

A GenVoca model of a SPL is a set of base programs and features (transformations) that extend or elaborate programs. 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 features are transformations (unary functions):

```
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 [7][45]. Tools that validate feature compositions are discussed in [6][47]. 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 [10], and rewrite rules in program transformation systems [11] — 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 as well.

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. Non-terminal features correspond to GenVoca expressions.

3.2 A GenVoca 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, we use subscripted letters. M_0 is the base program of `MapStats`, $M_1 \ldots M_n$ are the (unary function) features of the MapStats feature diagram and $C_0 \ldots C_m$ 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:

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

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:

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

Each `MapStats` feature can encapsulate SVG and JavaScript refinements (cross-cuts) 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 [9], and in particular, the XAK tool.

XAK is both a language to refine XML documents and 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 the appending of child nodes.

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`' specifies a pointcut as an XPath expression, which in this case looks for nodes called '`mynode`'. The '`xr:append`' 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= ...
  <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 manually both JavaScript and JavaScript refinements, and used XAK to compose them. The resulting JavaScript program was produced by stripping XML tags.

1. Aspects can be implemented by transformations; aspect compilers transform an input program to a “woven” program where additional code has been appropriately inserted [36].

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 MDE to save us effort: we created a high-level *Domain-Specific Language (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 an XPath expression (pointcut) 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 [14]).

```
<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>
```

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 transformations 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 chart DSL specifications to be 4-10 times shorter than their generated JavaScript counterparts.[2]

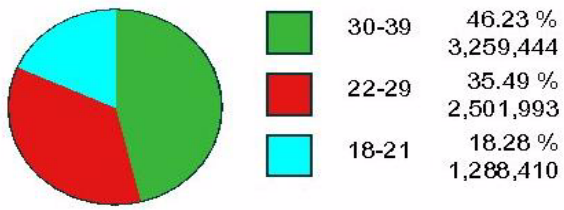


Fig. 8. Pie Chart with Three Age Categories

2. Our choice of using XML was due to the fact that SVG is an XML dialect, and using XSLT was an obvious choice for a transformation language at the time. After our work on `MapStats`, better model representations and transformation languages have emerged (e.g. [28]). Our results are general and not specific to XML and XSLT.

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:

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

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:

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

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

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.[3]

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 applies 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):

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

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. Operators (like τ and τ') are used to define maps between lifted features and their unlifted counterparts. Constraints that govern the composition of original MapStats features remain unchanged.

5 Implementation

In this section, we illustrate some of the features and mappings discussed earlier 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 and 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:

---

3. τ is really a special case of τ'. For this paper, we find it useful to distinguish τ from τ'.

```
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 us 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`. 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 that is double-underlined 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 double-underlined code 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 double-underlined code of `(5)` is generated by instantiating the τ' template with these parameters. By writing a general operator τ' once and reusing it (to translate other `Charts` features that were differentiated only by their parameters) saved us considerable effort. 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, adding features S_1 and S_2, and *then* transforming S into its corresponding JavaScript implementation. That is, another way to synthesize application A is:

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

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

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

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 its implementation is demonstrated when its commuting relationships are shown to hold. Commuting relationships provide a simple means to express 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.[4]
>
> **Note**: The justification for (8) follows intuition. Each S feature corresponds to a Charts feature. Each S feature composition corresponds to a unique Charts feature composition. τ relates S values to MapStats values, and τ' relates S functions to MapStats functions. When the equalities of commuting diagrams do not hold, we know there are bugs in our transformation or tool chains [49][50].

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 test cases. Other tests were performed with randomly selected features to ensure that each properly transformed the ap-

---

4. We differentiate τ and τ' because of their different operands: values vs. functions. Instead of writing (8), we could use the more general relationship (9) where Φ denotes both τ and τ'.

propriately selected features. Although more sophisticated and thorough testing was possible (see [37]), 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 [50], and over a factor of 10 in synthesizing test programs using Alloy [33]. Although we did observe trade-offs in building MapStats applications, they were not significant. The utility of commuting relationships in MapStats was restricted to testing model and transformation correctness.

7 Other Examples of Lifting

Our MapStats work clarified the role of lifting in FOP, but our earlier implementation of the *AHEAD Tool Suite (ATS)* contains other examples of lifting in rougher form. In retrospect, we see that the ATS implementation highlights issues that should be addressed if lifting is to be successful in FOP. We describe some of the issues from ATS in this section and their associated problems.

Lifting in FOP is analogous to the manual development of a generator (τ operator) for a *higher-level representation (HLR)*, just as feature composition itself is analogous to *program development by stepwise refinement* [53]. When developing ATS, we explicitly showed the stepwise refinement analogy by bootstrapping ATS from its own feature set, wherein each feature represented a refinement. However, FOP is more than just the automatic composition of refinement steps encapsulated into features. Given appropriate tools, we created entire software product lines from the development of a set of features, i.e., features can be mixed and matched to yield all the variations in a software product line.

When we consider lifting in FOP, we see a similar analogy and similar gains. Even without the use of FOP, lifting to a HLR can be useful, as has been demonstrated by DSLs in key niches (e.g., relational databases, logical modelling, signal processing). A DSL compiler in these domains is analogous to a τ operator. Taking one more step, an appropriately defined HLR can support features with a feature composition operator, gaining the benefits of a SPL in HLR form. Taking the last step, where a τ' operator is defined to map individual features to a lower-level form, we gain the ability to check correctness of commuting diagrams, as described in Section 6, and the ability, again with appropriate tools, to individually test features (e.g., in a simulated environment, or in a test framework where other features are mocked).

Nonetheless, it should be emphasized that lifting in FOP is subject to the same cost constraints that arise in all of software engineering. To achieve gains, additional work is required: the τ and τ' operators must exist or be developed, as do the composition operators in both HLR and lower-level forms. A development team can justify the costs only if the additional gains reduce the long-term maintenance costs by at least as much.

7.1 Grammars

ATS implements a language and tools to support feature-based development of SPLs. ATS is boot-strapped — the tools of ATS are synthesized by composing features, so ATS is used to build itself. The main language of ATS is *Jakarta* (or Jak for short), which is a superset of Java 2 (i.e., Java without generics). In effect, Jakarta is a product line of Java dialects where new language constructs can be added to Java [5]. We expressed the set of Java dialects that can be produced by ATS as a GenVoca product line:

```
J =   {  Java,          // base preprocessor
         Ast,           // adds code quote, unquote
         Gscope,        // adds hygienic macros
         Sm,            // adds state machines
         CompJava,      // adds refinements to Java classes, interfaces
         CompSm,        // adds refinements to state machines
         …
      }
```

The lone base program is the Java preprocessor (`Java`), and the remaining features add constructs to the Java language, such as code quote and unquote (`Ast`) [5], a form of hygienic macros called *generation scoping* (`Gscope`) [46], state machine declarations (`Sm`), adding refinements to Java classes and interfaces (`CompJava`), and adding refinements to state machines (`CompSm`) [10]. The base feature (`Java`) defined a complete Java language; the remaining features extend this language. Particular dialects of Java were created by composing features. For example, the dialect of Java that supports code quote/unquote and generation scoping is:

$$\texttt{Jakarta}_1 = \texttt{Gscope•Ast•Java}$$

We can extend this dialect to additionally support state machines, and class, interface, and state machine refinements:

$$\begin{aligned}\texttt{Jakarta}_2 &= \texttt{CompSm•CompJava•Sm•Jakarta}_1 \\ &= \texttt{CompSm•CompJava•Sm•Gscope•Ast•Java}\end{aligned} \tag{10}$$

An ATS build works in the following way: a language dialect, like $\texttt{Jakarta}_2$ above, is specified, and the build process generates a number of tools to manipulate, analyze and transform programs written in that dialect. Among the tools that are produced:

- `jak2java` — a tool that translates a Jak program to its Java counterpart,
- `reform` — a tool that pretty-prints Jak programs, and
- `mmatrix` — a tool that harvests declarations from Jak programs.

Each feature of the `J` product line extends the Java language with some unique construct and is implemented by a pair of smaller features `(parser, code)`. For example, the `Java` feature is comprised of a parser of the Java language $\texttt{parser}_{\texttt{java}}$, and $\texttt{code}_{\texttt{java}}$ is the code that implements the semantic actions for each Java language production. Similarly, the `Ast` feature is defined by $\texttt{parser}_{\texttt{ast}}$, the extension to the Java parser that parses code quote/unquote constructs, and $\texttt{code}_{\texttt{ast}}$ that implements the semantic actions of these constructs, and so on.

In effect, each feature F_i is a composition of parser P_i and code C_i subfeatures. The parser and code subfeatures are commutative: exactly the same program results for either composition order as each updates or adds non-overlapping pieces of code to an ATS tool:

$$\begin{array}{lll} Java & = parser_{java} \bullet code_{java} & = code_{java} \bullet parser_{java} \\ Ast & = parser_{ast} \bullet code_{ast} & = code_{ast} \bullet parser_{ast} \\ Gscope & = parser_{gscope} \bullet code_{gscope} & = code_{gscope} \bullet parser_{gscope} \end{array}$$

In general:

$$F_i = P_i \bullet C_i = C_i \bullet P_i \tag{11}$$

Moreover, when different features are composed, the order in which code and parser subfeatures are composed is preserved (i.e., code subfeatures do not commute, parser subfeatures do not commute, but code and parser subfeatures commute):

$$F_i \bullet F_j = P_i \bullet C_i \bullet P_j \bullet C_j = P_i \bullet P_j \bullet C_i \bullet C_j = C_i \bullet C_j \bullet P_i \bullet P_j \tag{12}$$

Writing the semantic actions (`code`) for each feature was not difficult. But writing the `parser` for a feature was hard: this is where parser tools are needed. That is, instead of writing the code for a parser (or parser refinement), we defined a grammar (or grammar refinement) in terms of a DSL, called Bali, that uses a BNF-like syntax to define the tokens and productions of a language [5]. In effect, we lifted parsers and their extensions to a Bali specification where defining grammars, grammar extensions, and composing grammars with their extensions was easy. Just as we lifted MapStats charts features to Chart DSL specs/models, we lifted ATS parsers to Bali grammars.

Each Bali grammar is in 1-to-1 correspondence with an ATS parser, and each Bali grammar extension is in 1-to-1 correspondence with an ATS parser extension. Recall in MapStats the τ transformation mapped a complete Charts spec to a MapStats function. The τ transformation for Bali was a tool chain: `bali2javacc` followed by JavaCC [27]. The `bali2javacc` tool transformed a Bali grammar to a JavaCC specification. The JavaCC tool was then used to transform the JavaCC specification into a Java parser.

What we did not have was a τ' operator, which would map an *extension* to a Bali grammar to an *extension* of a Java parser. At the time that we built ATS (circa 2001), we were unaware of existing compiler-compiler tools that could do this. We realized that with a τ' operator, the following homomorphism would hold:

$$\tau(G_i \bullet G) = \tau'(G_i) \bullet \tau(G) \tag{13}$$

where `G` denotes a complete grammar (i.e., a "value") and G_i denotes a grammar refinement (i.e., a "function"). As we did not have a τ' transformation, we could not compose features in the obvious way (e.g., `(10)`). Instead, we were forced to compose all grammar subfeatures first to produce a complete grammar for the target Java dialect, use this grammar to create a parser (using the τ transformation), and lastly compose the corresponding code subfeatures to build an ATS tool. Our ability to do this required the homomorphism of `(13)`.

To explain the process, let T be an ATS tool, and let its GenVoca expression be $F_2 \bullet F_1 \bullet F_0$. Let F_i denote any of these features, and let P_i and C_i denote F_i's parser and code subfeatures. Further, let B_i denote the Bali specification of P_i where:

$$P_0 = \tau(B_0)$$
$$P_i = \tau'(B_i) \quad \text{// for } i>0 \qquad (14)$$

It follows that:

$$\begin{aligned}
T &= F_2 \bullet F_1 \bullet F_0 && \text{// given} \\
&= P_2 \bullet C_2 \bullet P_1 \bullet C_1 \bullet P_0 \bullet C_0 && \text{// (11) substitution} \\
&= C_2 \bullet C_1 \bullet C_0 \bullet P_2 \bullet P_1 \bullet P_0 && \text{// (12) commutativity} \\
&= C_2 \bullet C_1 \bullet C_0 \bullet \tau'(B_2) \bullet \tau'(B_1) \bullet \tau(B_0) && \text{// (14) lifting} \\
&= C_2 \bullet C_1 \bullet C_0 \bullet \tau(B_2 \bullet B_1 \bullet B_0) && \text{// (13) homomorphism}
\end{aligned}$$

We used the last expression to implement the makefile of our ATS tool build process. Note that the reason why our makefile worked was because of the validity of (13), although only now do we have and understand a detailed explanation for its correctness.

7.1.1 Creating a τ' Operator for Grammars

Although we did not create a τ' operator for grammar extensions in ATS, we uncovered some guidelines for feature-oriented lifting.

First, we saw that the use of JavaCC, an LL(k) parser-generator, placed limitations on grammar extensions. It is easy to define a base grammar and a grammar extension that are individually LL(k), yet fail to be LL(k) when composed. A similar problem exists had we used an LR(k) parser-generator instead.

Example. Consider the elementary base grammar with start symbol "stmt":

```
stmt := "if" stmt "then" stmt
      | "other"
```

It has three terminals ("if", "then", and "other"), and the grammar is LL(1). Now consider the grammar extension (just one more production):

```
stmt := "if" stmt "then" stmt "else" stmt
```

This adds one more terminal, "else". More importantly, the combined grammar (base+extension) is no longer LL(1) because two different productions begin with the same terminal, "if". Further, the combined grammar is now ambiguous because it introduces the dangling-else problem. Specifically, consider input sentence:

```
if other then if other then other else other
```

There are two valid parse trees for this input. One where the "else" is bound to the first "if", and the second where the "else" is bound to the second "if". Finally, as the combined grammar is ambiguous, it is neither LL(k) nor LR(k), because those are defined to have unique parse trees for every valid language sentence.

In general, this type of problem can occur in feature compositions. As a guideline for implementors of paired composition and τ operators, the τ operator should have a domain space that is a superset of the range space of the composition operator. That is, it

is not sufficient to transform and compose individual features τ′(A) and τ(B), but τ must also be able to transform compositions of features, i.e., τ(A•B). It is possible for software developers to work around this issue by refactoring the grammars of A and B so that their composition is transformable by τ (as we did in ATS), but, since this requires additional manual effort, there are practical limits to this approach. (As an aside, we note that MapStats required similar restructuring as well).

In particular, the most popular parser-generators today are LL(k) or LR(k). If we implemented our guideline, our practical choices would have been among the parser-generator algorithms for ambiguous grammars, such as *Generalized LR (GLR)* [48] or Earley's algorithm [21]. Considering these two options highlights another guideline, which we'll call the "progress" guideline: Any τ' operator should make significant progress when translating from a HLR to a lower-level form. To make this clear, we contrast the use of GLR to Earley's algorithm.

The GLR algorithm, like LR, uses static state transition tables built from a given grammar. The distinction between GLR and LR occurs for shift-reduce or reduce-reduce conflicts: GLR effectively follows all possible choices, while LR either fails or uses an external directive to make a fixed choice. What is important to note here is that the entire grammar must be available in order for the state tables to be generated. As a corollary, any τ' operator acting on a grammar extension alone cannot be guaranteed to generate any significant part of the state tables.

By contrast, Earley's algorithm is dynamic. It uses a combination of an input program with an internal representation of grammar productions to build parse states dynamically. It is only necessary for a τ' operator to build the internal representation of a grammar extension's productions, which is significant progress in terms of Earley's algorithm. One could argue that a τ' operator for GLR could use the same production-level internal representation to simply encode each grammar extension, then leave it to a post-processing step to build the state transition tables. That would lead to a correct set of composition, τ, and τ' operators, but the τ' operators would not be computing a significant proportion of the state tables.

7.2 Bytecodes

Another example of lifting features in ATS deals with Java bytecodes. As mentioned earlier, most ATS tools are preprocessors. A typical ATS software development cycle is to (a) create the source code for features, (b) compose features (to produce composite source), and then (c) compile the composite source. This form of development is adequate for a research-based prototype, but is unacceptable for commercial development. One problem stands out: protection of intellectual property. Companies protect their intellectual property by distributing bytecodes, not source code. As the popularity of feature-based development increases, there will be an increasing demand for bytecode distribution, not source code distribution, of features.

Source code is a "lifted" representation of bytecode; source code is much easier to understand and write. The tools that we have today for mapping source code to bytecode are compilers (e.g., javac). These are our τ operators (i.e., they map a source code "value" to a bytecode "value"). What is missing is a τ′ operator that maps transformations

on source code (e.g., code modifications that a feature makes) to corresponding transformations on bytecodes.

In 2003, we prototyped a τ' operator that mapped the Jak source refinements of a feature to its corresponding representation as a bytecode refinement. We used a variation of a technique that was pioneered in Hyper/J for compiling hyperslices (i.e., Hyper/J modules) [40]. Basically an AHEAD feature is a hyperslice. To compile a hyperslice, stubs are created for all classes and members that are not introduced by that hyperslice. This makes them declaratively complete. Once stubs are available, the Java classes of a hyperslice can be compiled into bytecode. Hyper/J then uses bytecode composition tools to compose independently compiled hyperslices. We followed a similar approach.

We know of no tool support for automatic stub creation in Hyper/J; stubs must be created manually [35]. An advantage of AHEAD and product lines in general is that the source or binaries for all features are available. By analyzing the feature code base, we can automatically generate stubs for all classes that could appear in a synthesized product [9]. For every class, we created a stub that contains the union of the signatures of all fields, methods, and declarations that could appear in that class. The same for interfaces. Remember a Java class `C` in feature `F` encapsulates a fragment of a class `C` that could appear in a synthesized program `P`. When we compile feature `F`, we bind all references in class `C` of `F` to the fields, methods, and classes of our generated stubs. Only at feature composition time do we rebind each field, method, etc. reference in `C` of `F` to a definition of a field, method, or class that is supplied by the features that comprise `P`.

With our implementation of τ, we were able to demonstrate the following homomorphism:

$$\tau(C_i \bullet C) = \tau'(C_i) \bullet \tau(C) \qquad (15)$$

where C denotes a complete Java program (i.e., a "value") and C_i denotes a refinement of a Java program (i.e., a "function"). $\tau(C)$ denotes the bytecode of program C, and $\tau'(C_i)$ denotes the bytecode refinement of source code refinement C_i. Thus, we could produce the bytecodes of a program by either composing source code features and compiling, or by compiling source code features into bytecodes and composing bytecodes. We demonstrated to ourselves the correctness of (15) by synthesizing ATS tools both ways, and having both sets of tools pass the regression tests of ATS.

These experiments demonstrated that bytecode representation of features, and the ability to compose bytecode features, was indeed feasible. In doing so, we realized the limitations of our τ' tool. `javac` optimizes bytecodes, such as constant folding, constant propagation, and constant inlining. The problem here is that if two different features declare different values for the same constant, our approach to separate feature compilation would not work. We were fortunate that this situation didn't arise in the features of ATS. A general technology for τ' would postpone optimizations like constant folding and constant propagation until bytecode composition time, when all information about a target program and its features are known. Similarly, the way that we were able to compile features separately was not particularly elegant. What we needed was strong separate compilation [1][20][34], where class definitions can be compiled separately and where it is only in later stages of composition or loading that externally visible types

and values are resolved and optimized. Strong separate compilation has been previously proposed for Java for other reasons, and FOP only strengthens the arguments for its use. In general, strong separate compilation is an example of another guideline for FOP and feature-oriented lifting: the τ, τ', and composition operators should delay the binding of linking symbols (those used to specify compositions) until the final generation step.

What our experiment showed is that we could *partially* demonstrate the correctness of `(15)`: there were features that we could not correctly and separately compile to bytecodes (e.g., features that assigned different values to constants). Once again, we learned that common software development tools define the value-to-value mappings of liftings, but do not provide the function-to-function mappings of operators that liftings require. In this case, a robust τ' operator will be essential for feature-based program development to transition to industry, as bytecode distribution of features, rather than source code distribution of features, will be demanded.

7.3 Other Examples

We have another result where operators (i.e., mappings of transformations to transformations) were used, and this result also could be cast as an example of lifting. The application is to create a MDE product line of portlets [50]. Rather than writing low-level Jak and JSP source code, we lifted the problem into a DSL for defining state charts, state chart features, and their compositions. A state chart feature extends a base state chart and can be translated into a corresponding extension of low-level Jak and JSP source code. This exposes a homomorphism which allows state charts and their refinements to be composed, and then translated to Jak and JSP code, or equivalently, state charts and their refinements are separately translated to Jak and JSP code, and the resulting source is composed. We explain in [50] how τ' operators were created, and how we used homomorphisms to both optimize the synthesis of Jak and JSP code, and validate our implementations.

8 Related Work

FOP and MDE paradigms have their historic roots in Lisp, which promoted the concept that programs are values (or “programs as data”) and transformations are functions that map values to values. Hemel shows that model-to-model transformations are the same as model-to-code transformations where the target model has a meta-model of the code representation [26]. Other literature uses the concept of applying changes to models through transformations, which is fundamental to features [15][25][26][41][42][52].

Combining MDE and product line transformations is not new [2][4][17] [44][49][50]. Trujillo et al. used XAK and AHEAD to build web portlets from state chart models [50][51]. 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. Also, 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 [49].

Avila-García 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 have proposed using MDE to help product line engineers determine application variation points, and to assess the feasibility of automating software product line development with MDE [23]. Deelstra used MDE as a means of identifying variations points within a product line [19]. Both papers infer that a feature could use *Platform Independent Model (PIM)* to *Platform Specific Model (PSM)* transformations to implement different platforms and implementation technologies.

Czarnecki and Helsen combined features and MDE in a different way by surveying different types of transformation methods and analyzing the various features of these methods [18]. Other prior work defined a taxonomy of different types of transformations and classified them as endogenous and exogenous [38]. 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 [17]. 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 [32]. The design of web applications includes functionality, content, navigation and presentation components. Gronmo worked with cross-cutting aspects that are applied to sequence diagrams [25]. Advice is much like method-extensions in features with the difference that advice can have more than one joinpoint.

Adding exogenous transformations, which convert models from one metamodel to another, adds more complexity to the transformation process. Rath studied live updating of transformation outputs as the input is incrementally changed [41]. The application of features is much like changes to the input model in their work. Features of a source model must be transformed to features of the target models.

Many results in MDE have laid a foundation for model transformations [12][13][30]. 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. In the ATS case, the Bali grammar formed a PIM and the Java parser formed a PSM.

What differentiates our work is our use of elementary mathematics as a language to express our generative/MDE designs. 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.

9 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 implemented 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 MDE 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 testing that these properties hold helps demonstrate model correctness. Further, we noted that lifting is a general technique for MDE product lines. Our additional case studies involving ATS demonstrated this.

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.

10 References

[1] Ancona, D., Lagorio, G., Zucca, E.: True Separate Compilation of Java Classes. In: *Principles and Practice of Declarative Programming (PPDP'02)*, pp. 189-200. Pittsburgh, PA, USA (2002).

[2] Anastaspoulos, M., Forster, T., Muthig, D.: Optimizing Model Driven Development by Deriving Code Generation Patterns from Product Line Architectures. In: Hirschfeld, R., Kowalczyk, R., Polze, A., Weske, M. (Eds.): *Net.ObjectDays*, pp. 425-438. Erfurt, Germany (2005).

[3] Anfurrutia, F.I., Diaz,O., Trujillo, S.: On Refining XML Artifacts. In: *Int. Conf. on Web Engineering*, pp. 473-478. Como, Italy (2007)

[4] Avila-Garcia, O., Garcia, A.E., Rebull, E.V.S.: Using Software Product Lines to Manage Model Families in Model-Driven Engineering. In: *ACM Symposium on Applied Computing*, pp. 1006-1011. Seoul, Korea (2007)

[5] Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: *Int. Conf. on Software Reuse*, pp. 143-153. Victoria, British Columbia, Canada (1998)

[6] Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: *Software Product Line Conference (SPLC)*, pp. 7-20. Rennes, France (2005)

[7] Batory, D., Robertson, E., Chen, G., Wang, T.: Design Wizards and Visual Programming Environments for Genvoca Generators. *IEEE Trans. on Softw. Eng. (2000)* doi:10.1109/32.846301

[8] Batory, D., O'Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. on Softw. Eng. and Methodology*, Vol. 1, Issue 4, pp. 355-398 (1992)

[9] Batory, D., Sarvela, J.N.: The AHEAD Tool Suite (ATS). Website: Department of Computer Sciences, The University of Texas at Austin. `http://www.cs.utexas.edu/users/schwartz/index.html` (2003-2009). Accessed 24 July 2009.

[10] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step Wise Refinement. *IEEE Trans. on Softw. Eng.* (2004) doi:10.1109/TSE.2004.23

[11] Baxter, I.: Design Maintenance Systems, *Commun. of the ACM* (1992) doi:10.1145/129852.129859

[12] Bezivin, J.: Model Driven Engineering: An Emerging Technical Space. *Generative and Transformational Techniques in Software Engineering*. LNCS Volume 4143/2006, pp. 36-44.

[13] Booch, G., Brown, A., Iyengar, S., Rumbaugh, J., Selic, B.: An MDA Manifesto. *The MDA Journal*, May 2004, pp. 2-9

[14] Colyer, A., Rashid, A., Blair, G.: On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, Computing Department, Lancaster University. `http://www.comp.lancs.ac.uk/computing/aose/papers/COMP-001-2004.pdf` (2004). Accessed 27 July 2009

[15] Cuadrado, J., Molina, J.: Approaches for Model Transformation Reuse: Factorization and Composition. In: *Int. Conf. on Theory and Practice of Model Transformations*, pp. 168-182. Zurich, Switzerland (2008)

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

[17] Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: *Int. Conf. on Generative Programming and Component Engineering (GPCE)*, pp. 422-437. Tallinn, Estonia (2005)

[18] Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. In: *IBM Systems Journal*, Vol. 45#3, pp. 621-645 (2006)

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

[20] Drossopoulou, S., Eisenbach, S., and Wragg, D.: A Fragment Calculus Towards a Model of Separate Compilation, Linking and Binary Compatibility. In: *Symposium on Logic in Computer Science* (1999)

[21] Earley, J.: An Efficient Context-Free Parsing Algorithm. In: *Communications of the ACM* 13#2, pp. 92-104 (1970)

[22] Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: *Fundamental Approaches to Software Engineering (FASE)*, pp. 72-86 (2007)

[23] Gonzlez-Baixauli, B., Laguna, M.A., Crespo, Y.: Product Lines, Features, and MDD. In: *European Workshop on Model Transformations (EWMT)* (2005)

[24] Gray, J., et al.: Model Driven Program Transformation of a Large Avionics Framework. In: *Int. Conf. on Generative Programming and Component Engineering (GPCE)*, pp. 361-378 (2004)

[25] Gronmo, R., Sorensen, F., Moller-Pedersen, B., Krogdahl, S.: Semantics-Based Weaving of UML Sequence Diagrams. In: *Int. Conf. on Model Transformations (ICMT)*, pp 122-136 (2008)

[26] Hemel, Z., Kats, L., Visser, E.: Code Generation by Model Transformation. A Case Study in Transformation Modularity. In: *Int. Conf. on Model Transformations (ICMT)*, pp 183-198 (2008)

[27] JavaCC. `https://javacc.dev.java.net.` Accessed 23 July 2009

[28] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: *Model Transformations in Practice Workshop* at *Model Driven Engineering, Languages and Systems (MODELS)* 2005

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

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

[31] Kolbly, D.: *Extensible Language Implementation*. Ph.D. Dept. Computer Sciences, University of Texas at Austin (2002)

[32] Kurtev, I., van den Berg, K.: Building Adaptable and Reusable XML Applications with Model Transformations. In: *Int. World Wide Web (WWW) Conf.* (2005)

[33] Khurshid, S., Uzuncaova, E., Garcia, D., Batory, D.: Testing Software Product Lines Using Incremental Test Generation. In: *Int. Symp. on Software Reliability Engineering (ISSRE)* (2008), pp. 249-258.

[34] Lagorio, G: *Type Systems for Java Separate Compilation and Selective Recompilation*. PhD thesis, University of Genova (2004)

[35] Lopez-Herrejon, R., Batory, D.: Using Hyper/J to implement Product Lines: A Case Study. Dept. Computer Sciences, University of Texas at Austin (2002)

[36] Lopez-Herrejon, R., Batory, D., and Lengauer, C.: A Disciplined Approach to Aspect Composition. In: *Partial Evaluation and Program Manipulation (PEPM)* (2006), pp. 68 - 77

[37] Memon, A.M., Pollack, M.E., Soffa, M.L.: Using a Goal-driven Approach to Generate Test Cases for GUIs. In: *Int. Conf. on Software Engineering (ICSE)* (1999), pp. 257 - 266

[38] Mens, T., Czarnecki, K., van Gorp, P.: A Taxonomy of Model Transformations. In: *Int. Workshop on Graph and Model Transformations (GraMoT)* (2005). doi:10.1016/j.entcs.2005.10.021

[39] Neuman, A.: US Population 2000: Ethnic Structure and Age Distribution. `http://www.carto.net/papers/svg/samples`. Accessed Apr 2006.

[40] Ossher, H., P. Tarr: Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In: *Software Architecture and Component Technology*, Kluwer, Boston (2002)

[41] Rath, I., Bergmann, G., Okros, A., Varro, D.: Live Model Transformations Driven by Incremental Pattern Matching. In: *Int. Conf. on Model Transformation (ICMT)* (2008), doi:10.1007/978-3-540-69927-9_8

[42] Rivera, J., Vallecillo, A.: Representing and Operating with Model Differences. In: Int. Conf. Objects, Models, Components, Patters (*TOOLS EUROPE) (2008)*. doi:10.1007/978-3-540-69824-1_9

[43] Sabetzadeh, M., Easterbrook, S. M.: Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretic Approach. In: *Automated Software Engineering (ASE)* (2003), pp.12

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

[45] Selinger, P., et al.: Access Path Selection in a Relational Database System. In: *ACM Special Interest Group On Management of Data (SIGMOD)* (1979), pp. 23-34

[46] Smaragdakis, Y., Batory, D: Scoping Constructs for Program Generators. In: *Generative and Component-Based Software Engineering (GPCE)* (1999)

[47] Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe Composition of Product Lines. In: *Generative Programming and Component Engineering (GPCE)* (2007), pp. 95- 104.

[48] Tomita, M.: *Efficient Parsing for Natural Language: a Fast Algorithm for Practical Systems*. Kluwer, Boston (1985)

[49] Trujillo, S., Azanza, W., Diaz, O.: Generative Metaprogramming. In: *Generative Programming and Component Engineering (GPCE) (2007)*, pp. 105-114 .

[50] Trujillo, S., Batory, D., Diaz, O.: Feature Oriented Model Driven Development: A Case Study for Portlets. In: *Int. Conf. on Software Engineering (ICSE)* (2007), pp. 44-53.

[51] Trujullo, S.: *Feature Oriented Model Driven Product Lines*. PhD thesis, The University of the Basque Country. (2007)

[52] Wagelaar, D.: Composition Techniques for Rule-Based Model Transformation Languages. In: *Int. Conf. on Model Transformations (ICMT)* (2008). doi:10.1007/978-3-540-69927-9_11

[53] Wirth, N.: Program Development by Stepwise Refinement. In: *Communications of the ACM*, Vol. 14 Num 4, pp. 221-227 (1971)