The Objects and Arrows of Computational Design

Don Batory[1], Maider Azanza[2], and João Saraiva[3]

[1] University of Texas at Austin, Austin, Texas, USA
batory@cs.utexas.edu
[2] University of the Basque Country, San Sebastian, Spain
maider.azanza@ehu.es
[3] Universidade do Minho, Campus de Gualtar, Braga, Portugal
jas@di.uminho.pt

Abstract. *Computational Design (CD)* is a paradigm where both program design and program synthesis are computations. CD merges *Model Driven Engineering (MDE)* which synthesizes programs by transforming models, with *Software Product Lines (SPL)* where programs are synthesized by composing transformations called features. In this paper, basic relationships between MDE and SPL are explored using the language of modern mathematics.

Note: Although jointly authored, this paper is written as presented by Batory in his MODELS 2008 keynote.

Keywords: Software product lines, model driven engineering, categories.

1 Introduction

The future of program design and development lies in automation — the mechanization of repetitive tasks to free programmers from mundane activities so that they can tackle more creative problems. We are entering the age of *Computational Design (CD)*, where both program design and program synthesis are computations [39]. By *design*, I mean "what are the steps to create a program that meets a specification" (i.e., do this, then this, etc.). Such a script is called a *metaprogram*. By *synthesis*, I mean "execute these steps to produce the program". This is metaprogram execution.

At the forefront of Computational Design are two complementary but different technologies: *Model Driven Engineering (MDE)* and *Software Product Lines (SPL)*. These technologies have much in common and may soon be hard to distinguish. But abstractly for this paper, I will refer to *"pure" MDE* as defining high-level models of an application, and transforming these models into low-level artifacts, such as executables. "Pure" MDE is a general paradigm for program synthesis. In contrast, I will refer to *"pure" SPL* as a domain-specific paradigm for program synthesis. It exploits the knowledge of problems in a particular domain, tried-and-tested solutions to these problems, and the desire to automate the construction of such programs given this knowledge. Both "pure" MDE and "pure" SPL are synergistic: the strengths of one are the weaknesses of the other. MDE and SPL are clearly not mutually-disjoint technologies, but I will present their strengths as such here.

In a prior lifetime, I was a database researcher. My introduction to program synthesis was *relational query optimization (RQO)* [32]. The design of a query evaluation program was defined by a composition of relational algebra operations, a.k.a. a relational algebra expression. Expressions were optimized by applying algebraic identities called rewrite rules. Applying rules was the task of a query optimizer. It took me years to appreciate the significance and generality of RQO: it is a compositional paradigm for program synthesis and is a classic example of Computational Design. RQO fundamentally shaped my view of automated software development more than any software engineering course (in the 1980s and maybe even now) could have.

My research focusses on SPLs, where the goal is to design and synthesize any member of a family of related programs automatically from declarative specifications. The thrust of my early work was on language and tool support for SPLs. More recently, my interest shifted to elucidate the foundational concepts of SPL and MDE. Early on, I needed a simple modeling language to express program design and synthesis as a computation. I discovered that modern mathematics fit the bill.

Here's the reason: software engineers define structures called programs and use tools to transform, manipulate, and analyze them. Object orientation uses methods, classes, and packages to structure programs. Compilers transform source structures into bytecode structures. Refactoring tools transform source structures into other source structures, and metamodels of MDE define the allowable structures of model instances: transformations map metamodel instances to instances of other metamodels for purposes of analysis and synthesis. Software engineering is replete with such examples.

Mathematics is the science of structures and their relationships. I use mathematics as an informal modeling language (*not as a formal model*) to explain Computational Design. Certainly I claim no contributions to mathematics, but I do lay claim to exposing its relevance in informal modeling in SPLs. The foundation of my work rests on ancient ideas: that programs are data or values, transformations map programs to programs, and operators map transformations to transformations [11]. This orientation naturally lead me to MDE, with its emphasis on transformations.

Table 1. MDE, SPL, and Category Theory Terminology

Paradigm	**Object**	**Point**	**Arrow**
MDE	metamodel	model	transformation
SPL		program	feature

The goal of this paper is to expose a set of concepts on which MDE, SPL, and Computation Design are founded. Although the concepts come from category theory [25][30], a general theory of mathematical structures and their relationships, this paper is aimed at practitioners who do not have a mathematical background. I show how MDE and SPL ideas map to categorical concepts, and throughout this paper, I explain the benefits in making a connection. Table 1 summarizes the terminological correspondence. Basic concepts of category theory are in use today, but I suspect members of the SPL and MDE communities may not appreciate them. Also, as this conference is about modeling, it is worth noting that mathematicians can be superb modelers, and leveraging their ideas is largely what this paper is about. I begin by explaining a simple relationship between MDE and categories.

2 MDE and Categories

In category theory, an object is a domain of points (there does not seem to be a standard name for object instances — I follow Lawvere's text and use 'points' [25]).[1] In Figure 1a, an object is depicted with its domain of points, shown as a *cone of instances*. This diagram is familiar to the MDE community as a metamodel and its model instances (Fig.1b). "Pure" MDE focuses on a particular technology implementation (e.g., MOF, Ecore) of metamodels and their instances. However, the ideas of objects and instances are more general. This observation has been noted by others, e.g., Bézivin's technical spaces [24] and GROVE [34]. So one can think of a Java "object" whose instances are Java programs, a bytecode "object" whose instances are Java bytecode files, an XML "object" (XML schemata) whose instances are XML files, and so on.

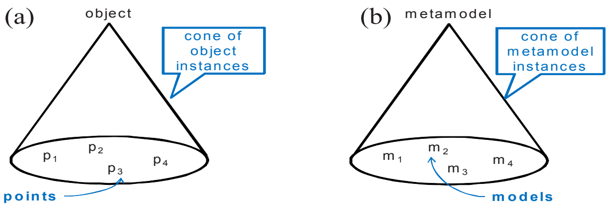


Fig. 1. Objects as Domains of points

Recursion is fundamental to category theory: a point can be an object. Fig. 2a depicts such a situation, which readers will recognize as isomorphic to the standard multi-level MOF architecture of Fig.2b:

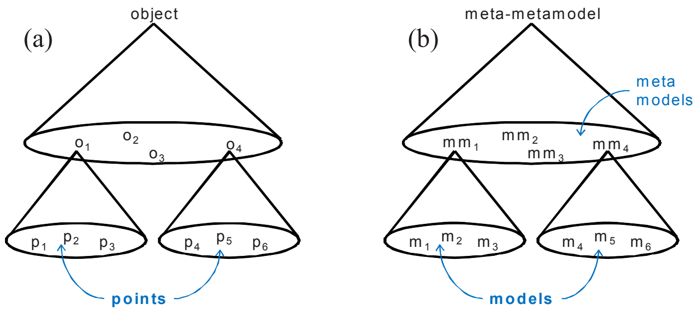

(a)
object
o_1 o_2 o_3 o_4
p_1 p_2 p_3 p_4 p_5 p_6
points
(b)
meta-metamodel
meta models
mm_1 mm_2 mm_3 mm_4
m_1 m_2 m_3 m_4 m_5 m_6
models

Fig. 2. Three-Level MOF Architecture

The MOF architecture is not interesting in category theory without arrows. An *arrow* is a *map* or *function* or *morphism* between objects, and whose implementation is unspecified.[2] Figure 3a shows an *external diagram* [25] that displays two objects, **S**

[1] I recommend Pierce's text on category theory [30] with concrete examples in [12] as illustrations; Lawvere's text is also quite accessible [25].

[2] A morphism is not necessarily a function; it can express a relationship, e.g., $\geq$.

and **J**, and an arrow **A** that maps each point of **S** to a point in **J**. (Arrows are always total, not partial, maps [30]). Figure 3b shows a corresponding *internal diagram* that exposes the points of every object of an external diagram and the mapping relationships among points. In general, there can be any number of arrows that connect objects; Figure 3c shows an arrow **B** that is different from **A**.

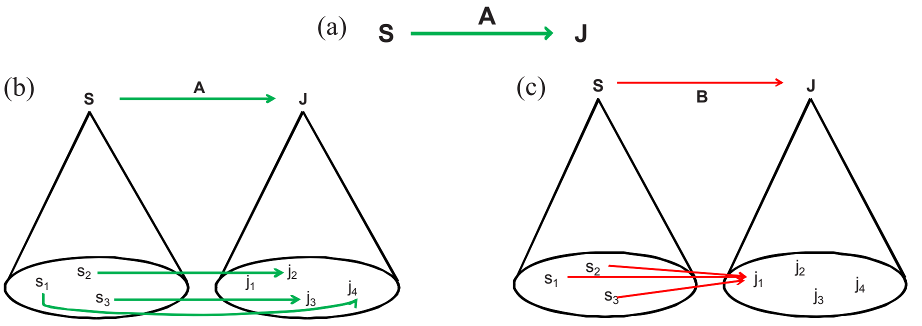


Fig. 3. External and Internal Diagrams

In this paper, I use the following terminology: an *arrow* is a mapping relationship, a *transformation* is an implementation of an arrow as expressed by an MDE transformation language (e.g., QVT [29], ATL [21], RubyTL [16], GReAT [1]), and a *tool* is any other (e.g., Java, Python) implementation of an arrow.

Now consider the following example: the external diagram of Fig. 4 shows that a message sequence chart (**MSC**) can be mapped to a state chart (**SC**) via a model-to-model transformation (**M2MX**). A state chart can be mapped to a Java program by a model-to-text transformation (**M2TX**). And a Java program is mapped to bytecode by the **javac** tool. Each arrow is realized by a distinct technology. In addition to these arrows, there are also identity arrows for each object.

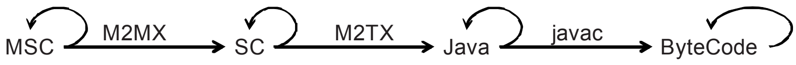


Fig. 4. Another External Diagram

There does not seem to be a standard name for such diagrams in MDE. Common names are tool chain diagrams [29] and megamodels [14] (both of which have slightly different graphical notations). Fig. 4 is also isomorphic to a UML class diagram, where metamodels are classes, transformations are methods, and fields of classes are private or hidden [10][34]. External diagrams are also standard depictions of categories. A *category* is a collection of objects and arrows, where each object has an identity arrow (i.e., identity transformation). Fig. 4 is an external diagram of a category of four objects and three non-identity arrows.

Besides objects and arrows, categories have the following properties [25][30]:

- Arrows are *composable*: given arrows $\mathbf{f:A}\to\mathbf{B}$ and $\mathbf{g:B}\to\mathbf{C}$, there is a composite arrow $\mathbf{g\bullet f:A}\to\mathbf{C}$.

- Composition is *associative*: given arrows **f:A→B**, **g:B→C**, and **h:C→D** (with **A**, **B**, **C**, and **D** not necessarily distinct), **h•(g•f) = (h•g)•f**.
- For each object **A**, there is an identity arrow $\mathbf{id}_A\mathbf{:A\to A}$ such that for any arrow **f:A→B**, $\mathbf{id}_B\mathbf{\bullet f = f}$ and $\mathbf{f\bullet id}_A\mathbf{=f}$.

Identity and composed arrows are often omitted from external diagrams.

The above properties allow us to infer that there is an arrow **T:MSC→ByteCode** that maps message sequence charts to Java bytecodes, where **T=javac•MT2X•M2MX** (**T** is not displayed in Fig. 4). In general, there needs to be tool support for these abstractions, so that all arrows, regardless on how they are implemented, can be treated uniformly. GROVE [34] and UniTI [37] are steps in the right direction.

Category theory defines arrows that map one input object to one output object. But in MDE, it is common in model weaving to map multiple models as input and produce multiple models as output [15]. Category theory has an elegant way to express this. The idea is to define a tuple of objects (called a *product of objects* [25][30]), and this tuple is itself an object. Projection arrows are defined so that each component of a tuple can be extracted. Fig. 5 depicts an arrow **F:[O1,O2,O3]→[O4,O5]** which maps a 3-tuple of objects to a pair of objects, along with projection arrows.

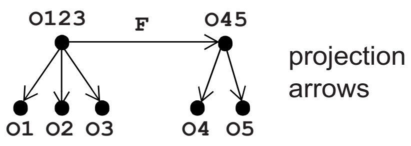


Fig. 5. Products of Objects

Now, let's look at Fig. 6, which depicts an internal diagram of Fig. 4. Although only one point is shown for each object, the relationships between these points ($\mathbf{m}_1$,$\mathbf{s}_1$,$\mathbf{j}_1$,$\mathbf{b}_1$) is also a category, sometimes called a *trivial category*, i.e., a category where each object represents a domain with a single point.

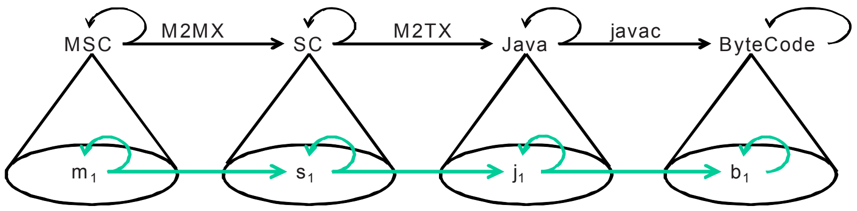


Fig. 6. An Internal Diagram of Fig. 4

In general, categories lie at the heart of MDE and can be found at all levels in a MDE architecture. Category theory provides an elegant set of ideas to express transformation relationships among objects that arise in MDE. The ideas are straightforward, if not familiar and have an elementary benefit: they may provide a clean foundation for MDE (e.g., such as a language and terminology to express MDE Computational Designs). A nice example of a formal use of categories in MDE is [19].

Now let's look at the connection between product lines and categories.

3 SPL and Categories

A software product line is a set of similar programs. Programs are constructed from features, which are increments in program functionality that customers use to distinguish one program from another. For example, program `P` is constructed from program `G` by adding feature `F`. This is expressed by modeling `F` as a function: `P=F(G)`.

The code of a product line of calculators is shown in Fig. 7. Associated with each line of code is a tag, which indicates the feature that adds that line. This makes it easy to build a preprocessor that receives as input the names of the desired features and strips off the code belonging to unneeded features. As can be imagined, this approach is brittle for problems of larger scale and complexity. Nevertheless, we use it as a reference to define what changes occur when a feature is added to a program.

```
base  class calculator {
base    int result;
base    void add( int x ) { result=+x; }
sub     void sub( int x ) { result=-x; }
base  }

base  class gui {
base    JButton add = new JButton("add");
sub     JButton sub = new JButton("sub");
form    JButton form = new JButton("format");

base    void initGui() {
base      ContentPane.add( add );
sub       ContentPane.add( sub );
form      ContentPane.add( form );
base    }

base    void initListeners() {
base      add.addActionListener(...);
sub       sub.addActionListener(...);
form      form.addActionListener(...);
base    }

form    void formatResultString() {...}
base  }
```

Fig. 7. Complete Code of the calculator SPL

The first program P_1 of the calculator SPL defines a `calculator` class and its `gui` class. The base calculator only allows numbers to be added. The second program P_2`=sub(`P_1`)`, extends the base with a subtraction operation (both the `calculator` and `gui` classes are updated). The effect of the `sub` feature is to add new methods and new fields to existing classes, and to extend existing methods with additional code. More generally, features can add new classes and packages as well. The third program P_3`=format(`P_2`)` adds an output formatting capability to a calculator, where again new methods and new fields are added, and existing methods are extended. A fourth program, P_4`=format(`P_1`)`, extends the base program with the `format` feature. One can push these ideas further, and say the base program is itself a feature, which extends the empty program `0`, i.e., P_1`=base(0)`. Feature `base` adds new classes (the base `calculator` class and the base `gui` class) to `0`.

These ideas scale: twenty years ago I built customizable databases (80K LOC each), ten years ago I built extensible Java preprocessors (40K LOC each), and more recently the AHEAD Tool Suite (250K LOC). All used the ideas that programs are values and transformations (features) map simple programs to more complex programs.[3]

Now consider the connection of SPLs to MDE. Fig. 8 shows a metamodel **MM** and its cone of instances. For typical metamodels, there is an infinite number of instances. An SPL, in contrast, is always a finite family of *n* similar programs (where *n* may range from 2 to thousands or more). So an SPL is a miniscule subset of a metamodel's domain. In fact, there is an infinite number of SPLs in a domain. If **MM** is a metamodel of state charts, it would not be difficult to find SPLs for, say, an IBM disk driver domain, a portlet flight booking domain, and many others.

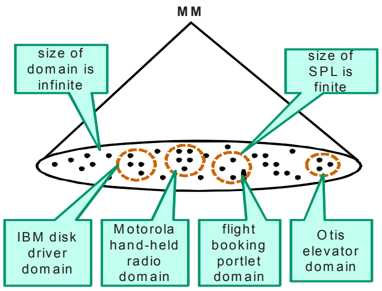

MM
size of domain is infinite
size of SPL is finite
IBM disk driver domain
Motorola hand-held radio domain
flight booking portlet domain
Otis elevator domain

Fig. 8. SPLs and Metamodels

As mentioned earlier, SPLs define relationships between its programs. How? By arrows, of course. Fig. 9 shows the calculator product line with its four programs, along with the empty program **0**, which typically is not a member of an SPL. Each arrow is a feature. From the last section, it is not difficult to recognize that an SPL is itself a trivial category: each point is a domain with a single program in it, there are implied identity arrows and implied composed arrows, as required.

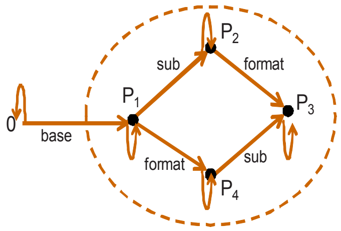


Fig. 9. Category of the Calculator SPL

Embodied in our description of SPLs is a fundamental approach to software design and construction, namely *incremental development*. Programs are built, step-by-step, by incrementally adding features. Not only does this control program complexity and improve program understandability, it also allows for the reuse of features (i.e., multiple programs in a product line could share the same feature). More on this shortly.

By composing arrows (features), the programs of an SPL are created. A program's design is an expression (i.e., a composition of arrows), and a program can have multiple, equivalent designs. For example, program $\mathbf{P}_3$ has two equivalent designs: **P**$_3$**=format•sub•base(0)** (which we henceforth abbreviate to **P**$_3$**=format•sub•base**) and **P**$_3$**=sub•format•base**. Evaluating both expressions yields exactly the same program. Features **sub** and **format** are said to be *commutative* because they modify mutually disjoint parts of a program.

3.1 Pragmatics of Software Product Lines

If there are *n* optional features, there can be 2^n different programs. We see a miniature example of this in Fig. 9: there are 2 optional features (**format** and **sub**) and there are

[3] Readers who are familiar with the decorator pattern will see a similarity with features: a decorator wraps an object to add behaviors. Features can be dynamically composed, but in this paper, they are statically composed to produce programs. Another difference is scale: decorators wrap a single object, whereas features often modify many classes of a program simultaneously.

2^2=4 programs in the product line. A slightly larger and more illustrative example is Fig. 10. We want to create an SPL of many programs; we know the arrows that allow us to build each of these programs, and many programs use the same feature, e.g., the thick arrows of Fig. 10 denote the application of the green feature to 4 different programs, and the dashed arrows denote the application of the blue feature to 3 different programs. It is the reuse of arrows that makes them more economical to store than programs.

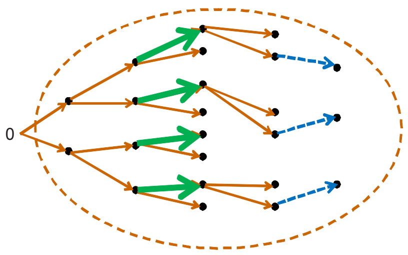


Fig. 10. Reuse of Arrows in SPLs

As an aside, features are like database transactions: they make changes to a program that are not necessarily localized: changes can appear anywhere in a program. But the key is that either all changes are made, or none are. Further, features can produce non-conforming programs (non-conforming models). Fig. 11 depicts an arrow **`F`** that relates programs $\texttt{P}_3$ and $\texttt{P}_6$, both of which conform to metamodel **`MM`**. But by arrow composibility, we see that **`F=F3•F2•F1`**. Applying **`F1`** to $\texttt{P}_3$ yields program $\texttt{P}_4$, and applying **`F2`** to $\texttt{P}_4$ yields program $\texttt{P}_5$, and $\texttt{P}_6\texttt{=F3(P}_5\texttt{)}$. Note that programs $\texttt{P}_4$ and $\texttt{P}_5$ do *not* conform to **`MM`**. It is common for existing features to be decomposed into compositions of smaller features, the individual application of which does not preserve conformance properties of the resulting program or model.[4] The reason why these smaller arrows arise is that features often have a lot of code in common. Commonalities can be factored into small features (small arrows) that are shared in implementations of larger arrows. We will see examples of small arrows in the next section.

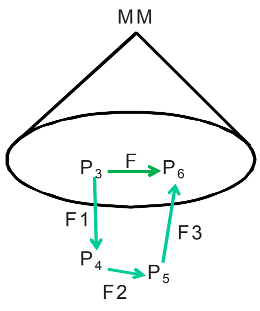


Fig. 11. **`F=F3•F2•F1`**

3.2 Arrow Implementations

There are two ways in which arrows are implemented. First is to implement arrows in the ATL, GReAT, etc. languages. The second and standard way for SPLs is that arrows are program or model *deltas* — a set of changes — that are superimposed on existing models (e.g., AHEAD [9], Scala [28], Aspectual Feature Modules [5], and AspectJ [22]). In effect, deltas can be viewed as a specific example of model weaving [15]. Which approach — writing modules that are to be superimposed or writing transformations— is "better"? This is not clear; I am unaware of any study to compare their trade-offs. In this paper, I focus solely on the use of deltas, so that core concepts in SPLs can be given their most direct MDE interpretation.

Here is an example. Fig. 12a shows the AHEAD representation of the **`sub`** feature of our calculator SPL. It states that the **`calculator`** class is extended with a "**`void sub(int x)`**" method, and the **`gui`** class is extended with a new field (**`JButton sub`**),

[4] Conformance for a program could be whether it type checks or not.

and its existing methods (`initGui()` and `initListeners()`) are wrapped (effectively adding more lines of code to these methods). We mentioned in the last section about decomposing a feature (arrow) into smaller features (arrows). Fig. 12b defines the AHEAD arrow (`subcl`) that changes only the `calculator` class, Fig. 12c defines the AHEAD arrow (`subgui`) that changes only the `gui` class. Composing `subcl` and `subgui` in either order produces `sub` (Fig. 12d). The `subgui` arrow could be decomposed even further, as a composition of arrows that introduce fields and wrap individual methods.

```
refines class calculator {
  void sub( float x ) { result=-x; }
}

refines class gui {
  JButton sub = new JButton("sub");

  void initGui() {
    SUPER.initGui();
    ContentPane.add( sub );
  }

  void initListeners() {
    SUPER.initListeners();
    add.addActionListener(...);
  }
}
```

(a) `sub=subcl•subgui=subgui•subcl`

```
refines class calculator {
  void sub( float x ) { result=-x; }
}
```

(b) `subcl`

```
refines class gui {
  JButton sub = new JButton("sub");

  void initGui() {
    SUPER.initGui();
    ContentPane.add( sub );
  }

  void initListeners() {
    SUPER.initListeners();
    add.addActionListener(...);
  }
}
```

(c) `subgui`

(d)

$$\begin{tikzcd}
 & {} \arrow[dr, "\mathtt{subcl}"] & \\
{} \arrow[ur, "\mathtt{subgui}"] \arrow[rr, "\mathtt{sub}"] \arrow[dr, "\mathtt{subcl}"'] & & {} \\
 & {} \arrow[ur, "\mathtt{subgui}"'] &
\end{tikzcd}$$

Fig. 12. AHEAD Arrow Implementations

The same ideas hold for MDE models. In two different product lines, fire support simulators [7] and web portlets [35], customized state machines were created by having features encapsulate fragments of state machines. By composing fragments, complete state machines were synthesized.

To me, the essential entities that programmers create in “pure” MDE are complete models (points); in “pure” SPLs they are features (arrows representing model deltas). Hence, there is discernible distinction between these paradigms, and exposing this distinction reveals an interesting perspective. Fig. 13a shows the metamodel `MM`, its cone of instances, and a particular product line `PL` whose members are $\mathtt{m}_1$, $\mathtt{m}_4$, and $\mathtt{m}_5$. The domain of a more general metamodel, called an *arrow metamodel* $\underline{\mathtt{MM}}$, is a superset of `MM`. Fig. 13b exposes the arrows that relate models in the `PL` product line, showing how models and features can be placed in the same cone of instances. Each model `m` is represented by an arrow $\mathtt{O}\rightarrow\mathtt{m}$. Fig. 13c erases `MM` and its cone to reveal that the instances of $\underline{\mathtt{MM}}$ are arrows. The subset of arrows that define `PL` is indicated in Fig. 13c, and so too are other sets of arrows (not necessarily disjoint) that are used to create other product lines. By combining a set of arrows with a feature model (i.e., a

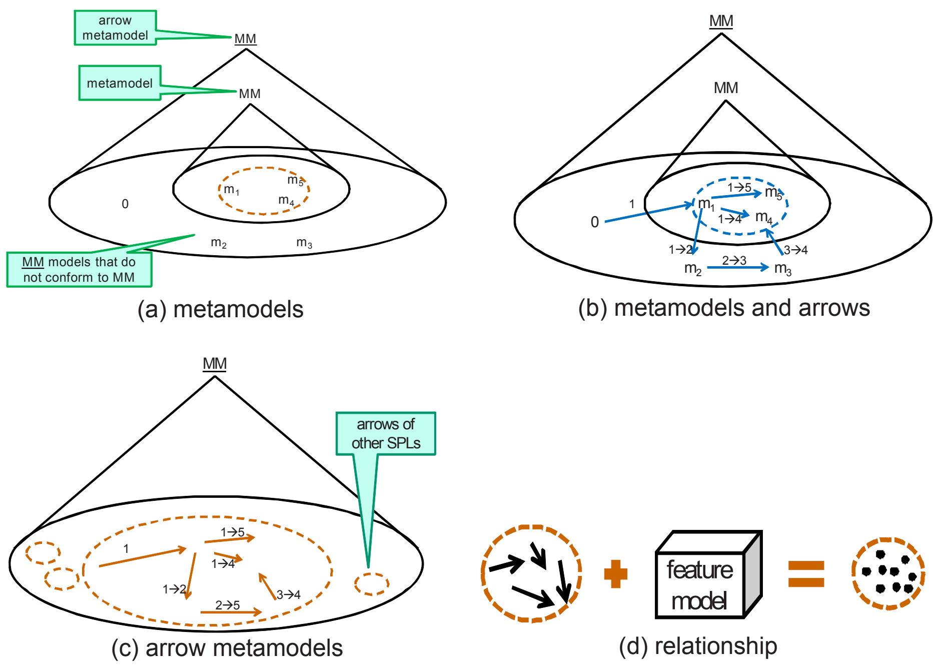


Fig. 13. Metamodels and Arrow MetaModels

specification that defines what composition of arrows are legal), the original product line **PL** within **MM**'s cone of instances can be generated (Fig. 13d).[5]

From a modeling perspective, the SPL approach to program construction recognizes a basic fact: all program artifacts — MDE models, Java programs, etc. — are not created spontaneously. They are created by extending simpler artifacts, and these artifacts come from simpler artifacts, recursively, until **0** is reached. The connection between successive artifacts is an arrow (from the simpler to the more complex artifact). By following arrows forward in time starting from **0**, an artifact (program, model, etc.) is synthesized. In effect, SPLs add a dimension of time to program or model designs. Proceeding forward in time explains how a program was developed in logical steps. Or stated differently, program synthesis is an integration of a series of well-understood changes.

3.3 Recursion

Product lines of models will be common, but product lines of metamodels, a form of product lines of product lines [8], will also be common. Fig. 14 depicts the MDE architecture. A product line of four metamodels is shown, along with the arrows that

[5] Support for deltas in conventional programming languages is largely absent. One can only define programs, *not* changes one wants to make to an existing program and encapsulate and compose such changes. It is as if one-half of a fundamental picture is absent.

connect them. Such arrows could be metamodel deltas (as we have described previously), or they could be refactorings [33][38]. Normally, when a metamodel is changed, one would like to automatically update all of its instances. The model-to-model transformation that is derived from a metamodel-to-metamodel transformation is called a *co-transformation* [33][38]. Co-transformations map product lines of one metamodel to product lines of other metamodels.

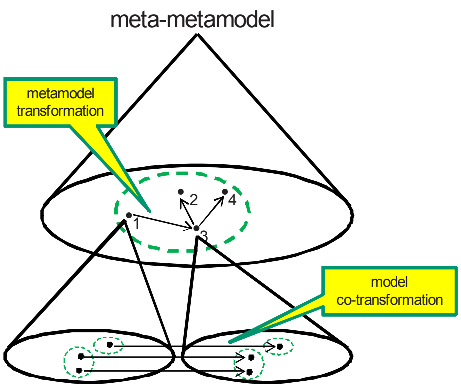


Fig. 14. Co-transformations

3.4 Recap

Categories lie at the heart of SPLs, and again the ideas are straightforward. Well-studied ideas in mathematics offers a clean language and terminology to express SPL Computational Designs. See [12] for an example. Now, let's see what happens when MDE and SPLs are combined into *model-driven software product lines (MDSPL).*

4 MDSPL and Categories

A fundamental concept in category theory is the *commuting diagram*, whose key property is that all paths from one object to another yield equivalent results. The diagram of Fig. 15a is said to commute if $f_2 \bullet d_1 = d_2 \bullet f_1$. Commuting diagrams are the theorems of category theory.

Commuting diagrams arise in MDSPL in the following way. Consider Fig. 15b, which shows arrow **A** that maps object **S** to object **B**. A small product line of **S**

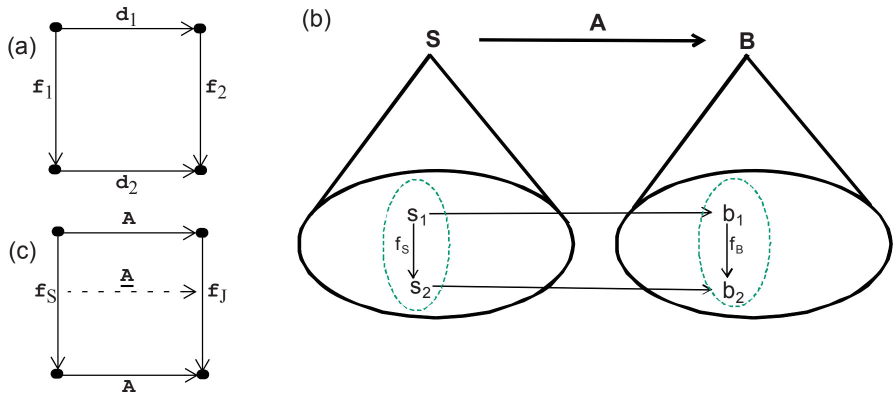


Fig. 15. Commuting Diagrams

instances is depicted, and these points are mapped by $\mathtt{A}$ to a corresponding set of points in $\mathtt{B}$. In general, horizontal arrows are MDE transformations and vertical arrows are SPL features. Note that feature $\mathtt{f}_S$ that relates $\mathtt{s}_1$ to $\mathtt{s}_2$ is mapped to a corresponding feature $\mathtt{f}_B$ that relates $\mathtt{b}_1$ to $\mathtt{b}_2$. Mapping a feature (arrow) to another feature (arrow) is an *operator* or update translation [4]. Operator $\underline{\mathtt{A}}$ relates features $\mathtt{f}_S$ and $\mathtt{f}_B$ in Fig. 15c by $\mathtt{f}_B\mathtt{=}\underline{\mathtt{A}}\mathtt{(f}_S\mathtt{)}$.

From our limited experience, operators can sometimes be easy to write; generally they pose a significant engineering challenge. As a challenge example, let $\mathtt{S}$ be the domain of Java source and $\mathtt{B}$ be domain of Java bytecodes. Suppose feature $\mathtt{f}_S$ is a delta in source code that maps the source $\mathtt{s}_1$ of program $\mathtt{P}_1$ to the source $\mathtt{s}_2$ of program $\mathtt{P}_2$. $\mathtt{f}_B$ is the delta that is applied to the binary of $\mathtt{s}_1$ to yield the binary of $\mathtt{s}_2$ (i.e., $\mathtt{b}_2\mathtt{=f}_B\mathtt{(b}_1\mathtt{)}$). Implementing operator $\underline{\mathtt{A}}$ requires separate class compilation, a sophisticated technology that can compile Java files individually and delay complete type checking and constant folding optimizations until composition time [2]. In the next sections, we present examples of operators we have implemented.[6]

Note: The generalization of metamodel $\mathtt{S}$ to the arrow metamodel $\underline{\mathtt{S}}$ as explained in Section 3.2 also applies to the generalization of arrows. That is, the external diagram consisting of objects $\mathtt{S}$ and $\mathtt{B}$ and arrow $\mathtt{A:S}\to\mathtt{B}$ can be generalized to the external diagram with objects $\underline{\mathtt{S}}$ and $\underline{\mathtt{B}}$ and arrow $\underline{\mathtt{A}}\mathtt{:}\underline{\mathtt{S}}\to\underline{\mathtt{B}}$. This is the $\underline{\mathtt{A}}$ operator discussed above.

Note: $\underline{\mathtt{A}}$ is a *homomorphism*: it is a mapping of $\underline{\mathtt{S}}$ expressions (compositions of one or more $\underline{\mathtt{S}}$ arrows) to a corresponding $\underline{\mathtt{B}}$ expression (compositions of one or more $\underline{\mathtt{B}}$ arrows). Let $\mathtt{x}$ and $\mathtt{y}$ be arrows of $\underline{\mathtt{S}}$. The commuting relationship of a homomorphism is:

$$\underline{\mathtt{A}}\mathtt{(x\bullet y)} = \underline{\mathtt{A}}\mathtt{(x)}\ \underline{\mathtt{A}}\mathtt{(\bullet)}\ \underline{\mathtt{A}}\mathtt{(y)}$$

where $\underline{\mathtt{A}}\mathtt{(\bullet)}$ typically maps to function composition (•). We talk about the practical benefits of such relationships next.

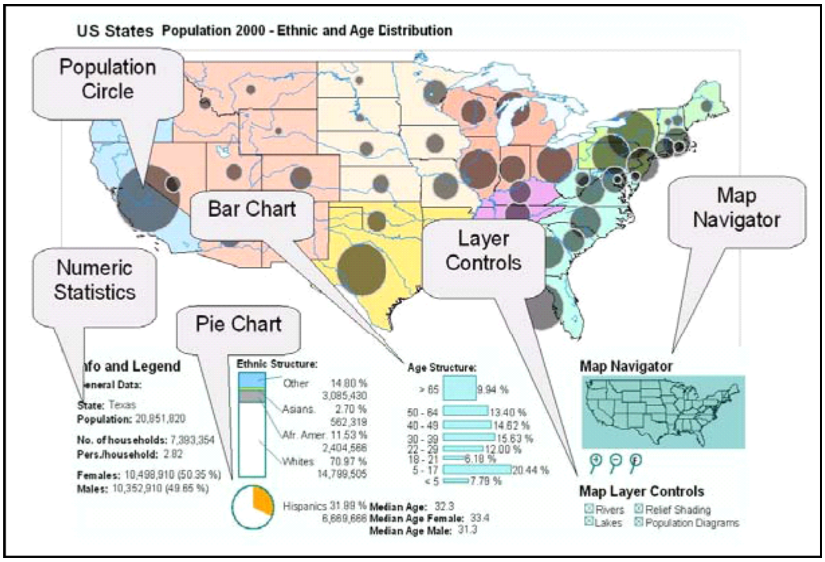
5 Benefits of Mapping Arrows

In the last two years, we discovered several uses for mapping arrows in MDE product lines: simplifying implementations [17], improving test generation [36], understanding feature interactions [23], explaining AHEAD [12], and improving the performance of program synthesis [35]. In the following sections, I briefly review two recent results.

5.1 Lifting

MapStats is an MDSPL where applications are written in SVG and JavaScript. MapStats applications display an SVG map of the U.S. where the map can be customized by adding or removing charts, statistics, and controls (Fig. 16).

[6] Gray has noticed that the kind of commuting diagrams shown here often require transformations that involve different technical spaces (using Bezivin's terminology). These are often hard to compose in practice, yet seem easy in these diagrams [18]. As mentioned earlier, there is a strong need for relating these tool chains [34][37].



nfo and Legend
eneral Data:
State: Texas
Population: 20,851,820
No. of households: 7,393,354
Pers./household: 2.82
Females: 10,498,910 (50.35 %)
Males: 10,352,910 (49.65 %)

Ethnic Structure:
Other 14.80 % 3,085,430
Asians 2.70 % 562,319
Afr. Amer. 11.53 % 2,404,566
Whites 70.97 % 14,799,505
Hispanics 31.99 % 6,669,666

Age Structure:
> 65 9.94 %
50 - 64 13.40 %
40 - 49 14.62 %
30 - 39 15.63 %
22 - 29 12.00 %
18 - 21 6.18 %
5 - 17 20.44 %
< 5 7.78 %

Median Age: 32.3
Median Age Female: 33.4
Median Age Male: 31.3

Fig. 16. A MapStats Application

The SPL design was a collection of MapStats features (arrows) and its feature model, which defined the legal combination of MapStats features. All MapStats features were implemented as XML documents or XML document deltas. By composing arrows using XAK, a general language and tool for refining XML documents [3], customized MapStats applications were synthesized. Early on, we discovered that a particular subset of arrows, namely those that implemented charts, were tedious to write. We used a basic concept of MDE to create a *domain-specific language (DSL)* to define charts and chart features. Each Chart feature was mapped to its corresponding and low-level MapStats feature by an operator (τ:<u>**`Chart`**</u>→<u>**`MapStats`**</u>). In effect, we "lifted" chart arrows from their MapStats implementation, to arrows in a Charts DSL (Fig. 17). By doing so, we simplified the writing of Charts arrows using the Charts DSL, and we automated the tedious implementations of their corresponding MapStats arrows.

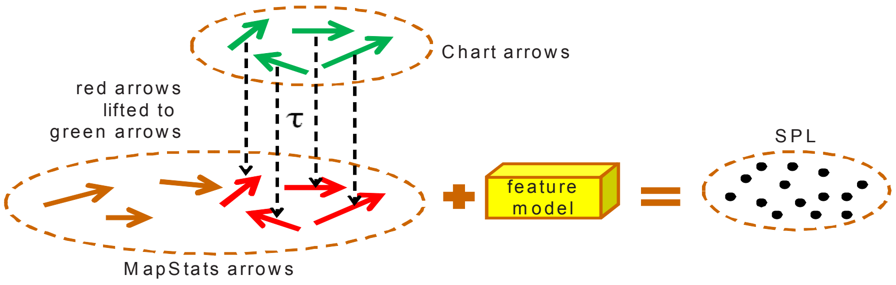


Fig. 17. Lifting Arrows

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

(a) chart specification

```
<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="blue" ...
    </xr:append>
  </xr:at>
</xr:refine>
```

(b) a Chart arrow

```
<chart data-type="age-population" type="pieChart" ...
  <item attr="AGE_30_39" color="green" name= ...
  <item attr="AGE_22_29" color="cyan" name=...
  <item attr="AGE_18_21" color="blue" name=...
</chart>
```

(c) a refined chart specification

```
<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("blue");
       </statement>
    </xr:append>
  </xr:at>
</xr:refine>
```

(d) corresponding MapStats arrow

Fig. 18. MapStats and Chart Arrows

As an example, Fig. 18a shows a simple DSL spec **S** of a pie-chart that displays age population for the ranges **30-39** and **22-29**. Fig. 18b shows a specification of a chart arrow **R** that adds the range **18-21** to a Chart spec. The underlined code defines a pointcut that identifies nodes in an XML document, and the advice is to append the **18-21** range item to selected nodes. Applying **R** to **S** (evaluating expression **R(S)**) yields the Chart spec of Fig. 18c. The τ operator maps a Chart arrow to a MapStats arrow. The result of τ**(R)** is the MapStats arrow of Fig. 18d. Note that τ maps the

Chart pointcut to the corresponding MapStats pointcut, and maps the Chart advice to the corresponding MapStats advice written in JavaScript. $\mathcal{T}$ was written in XSLT.

A homomorphism relates Chart arrows ($\mathtt{S}_k$) to MapStats arrows ($\mathtt{C}_k$):

$$\mathcal{T}(\mathtt{S}_i \bullet \mathtt{S}_j) = \mathcal{T}(\mathtt{S}_i) \bullet \mathcal{T}(\mathtt{S}_j) = \mathtt{C}_i \bullet \mathtt{C}_j \qquad (1)$$

We used (1) in two ways. First, when a particular MapStats application was specified as a composition of MapStats arrows, we used (1) to generate MapStats chart arrows. For example, let $\mathtt{M}_i$ denote non-chart arrows of MapStats. A MapStats application **P** is a composition of **M** arrows followed by **C** arrows. We translated **P** into equivalent expressions using (1) and evaluated either of these new expressions to synthesize **P**:

$$\begin{aligned}
\mathtt{P} &= \mathtt{C}_2 \bullet \mathtt{C}_1 \bullet \mathtt{C}_0 \bullet \mathtt{M}_1 \bullet \mathtt{M}_0 && \texttt{// given} \\
&= \mathcal{T}(\mathtt{S}_2 \bullet \mathtt{S}_1 \bullet \mathtt{S}_0) \bullet \mathtt{M}_1 \bullet \mathtt{M}_0 && \texttt{// by (1)} \\
&= \mathcal{T}(\mathtt{S}_2) \bullet \mathcal{T}(\mathtt{S}_1) \bullet \mathcal{T}(\mathtt{S}_0) \bullet \mathtt{M}_1 \bullet \mathtt{M}_0 && \texttt{// by (1)}
\end{aligned}$$

The second use of (1) was for verification: it defined a set of constraints that hold between pairs of Charts and MapStats features and their compositions. Here, as in previous experiences [35], our tools did not satisfy these constraints (meaning the equalities of (1) did not hold). This exposed bugs in our tools which we had to fix.[7] Now we have greater confidence in our tools as they implement explicit relationships in our MDSPL models. This is a win from an engineering perspective: we have insights into domains that we did not have before, and we have a better understanding, better models, and better tools as a result.

Lifting is a general technique that can be applied to many product lines. For more details, see [17].

5.2 Test Generation

Testing members of SPLs is a fundamental problem. We can synthesize different programs, but how do we know these programs are correct? In such cases, specification-based testing can be effective. Starting with a specification (or model) of a program, we want to derive its tests automatically. Alloy is an example of this approach [20].

Alloy works by translating an Alloy specification **S** into a propositional formula. A SAT solver finds the bindings that satisfy the formula, called a *solution*. Let **I** denote the set of all solutions for **S**. A test program is generated for each solution using the TestEra tool [26]. The set of all tests, **T**, is the output.

Alloy specifications can be developed incrementally by conjunction. That is, if program $\mathtt{P}_0$ has specification $\mathtt{S}_0$ and feature **F** has specification $\mathtt{S}_F$, then the spec of **F(P)** is $\mathtt{S}_0 \wedge \mathtt{S}_F$. The conventional way to synthesize tests for a program is to compose the specifications of all of its features, and then use Alloy and TestEra to produce its tests. We know there is a commuting diagram behind this design, which Fig. 19 exposes. The left column of objects are Alloy specifications, the middle column are spec solutions, and the right column are tests. Horizontal arrows are the tools $\mathtt{alloy{:}S}\rightarrow\mathtt{I}$ and $\mathtt{TestEra{:}I}\rightarrow\mathtt{T}$. Features are vertical arrows. The right-most column of vertical

[7] Although we could not prove the equivalence of (1), we could demonstrate equivalence by testing, as is done in conventional software development.

arrows are spec refinements. The middle column are solution refinements, and the right column are test refinements.

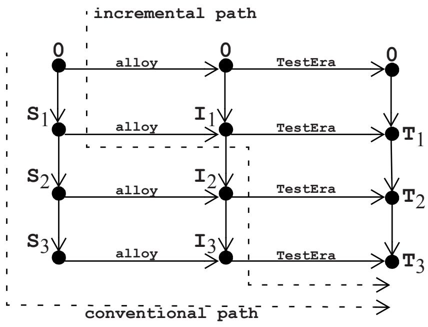


Fig. 19. Paths for Test Generation

Only the conventional path, and no other, has ever been taken. The challenge is to determine how to implement an operator τ:$\underline{\texttt{S}}\rightarrow\underline{\texttt{I}}$ to map spec arrows to solution arrows and maybe another operator σ:$\underline{\texttt{I}}\rightarrow\underline{\texttt{T}}$ to map solution arrows to test arrows. Uzun caova et al. discovered an elegant way to realize τ (for details, see [36]). This discovery exposed an alternative path, called the *incremental path*, that first derives the solutions for the base specification, and extends each solution to zero or more solutions of an incrementally more complicated specification. Once solutions to the target specification are found, the **TestEra** tool is used to produce the corresponding set of tests.

Initial experiments revealed that in a majority of cases, the incremental path synthesizes test programs faster than the conventional path, and for some cases, the speedup was 30-50× faster. Not surprisingly, other paths were found to be even more efficient (i.e., extend a specification multiple times, then derive its solutions, then extend these solutions). Of course, we know that there are test arrows that relate tests for different programs, but here is a case where it is unlikely that creating an operator σ to map solution arrows to test arrows would be useful — all the work in extending tests seems to be in extending solutions.

In general, commuting diagrams reveal new ways to solve problems, and in some cases, these new solutions are better than existing solutions.

5.3 Recap of Benefits

Exposing commuting relationships in program synthesis, as illustrated in the previous sections, has revealed a set of interesting problems and novel perspectives that have lead to useful results. I expect many more applications of commuting diagrams in the future. An even more interesting, longer-term, and open question is whether mathematicians can leverage this connection of MDE and SPLs to provide deeper results.

6 Design Optimization

Design optimization is the most exotic part of Computational Design. If a program's design is an expression, then the expression can be optimized to produce an equivalent and improved design. In the last section, we saw commuting diagrams offered different paths to produce equivalent results. In the case of test generation, finding the right path could shorten generation time substantially. There is a counterpart in SPLs which originates from relational query optimization, that I now briefly describe.

Relational query optimization makes a clean distinction between functional requirements and non-functional requirements. A functional requirement is an arrow (e.g., relational operation); a non-functional requirement is a computable, estimatable, or measurable property of a composition of arrows (e.g., performance) [32].

Fig. 20 depicts an SPL of multiple programs, all of which are derivable from `0`. A subset of these programs satisfy the functional requirements of a program spec. (This is the inner set of programs in Fig. 20). Designers want a program of this inner set that also satisfies non-functional requirements and/or optimizes some quality metrics (e.g., performance). In principle, by enumerating this inner set, evaluating each point on its quality behavior, and selecting the point that exhibits the "best" quality (e.g., most efficient program w.r.t. some criteria), that is the program to build. Of course, how one enumerates or searches the inner set, how one evaluates or ranks points on the basis of quality metrics, and to do so efficiently, is often a challenging engineering problem. But this is the RQO paradigm: each relational algebra operation is an arrow, relational algebra expressions are arrow compositions, and relational query optimization is expression optimization with respect to performance.

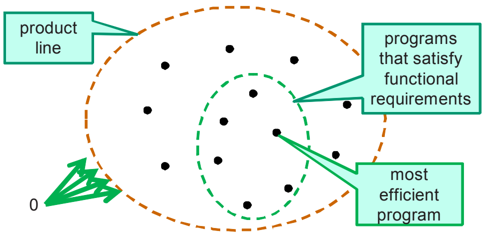


Fig. 20. Optimizing Program Designs

At present, I am aware of only a few examples of design optimization, among them are RQO [32], data structures [6], adaptive computing [27], middleware [40], and library synthesis [31]. A general technology for optimization may be constraint satisfaction [13]. The main challenge is finding domains where there are different ways of implementing the same functionality. Usually, most SPLs have only one implementation of a feature, and without multiple implementations, there may not be many opportunities for optimization a la RQO.

The key lesson is this: if you have a good conceptual framework, you will be able to recognize more easily the relationship among different and disparate areas of research. Much of what we do today as designers and implementors is to define and transform structures. By making these abstractions and distinctions clear(er), we will be that much closer to understanding the essence of MDE, SPLs, and Computational Design.

7 Conclusions

One of the key advances that brought database systems out of the stone age is relational query optimization. The relational model and the optimization of queries was rooted firmly in set theory, using elementary operations on sets (select, project, join, union). From a mathematical perspective, virtually nothing of set theory was used except for the first few pages in a set theory text. It was these simple ideas from set theory, not its deeper results, that made a lasting impact on databases.

The same may hold for category theory: its elementary ideas may find their way into the practice of MDE and SPL program design and synthesis. There is preliminary evidence that these ideas bring both pragmatic and pedagogical benefits. From an informal modeling viewpoint, the ideas I presented here are usable by engineers. Deeper results may be forthcoming.

How often will commuting diagrams arise in MDSPLs? This is not yet clear. One thing is clear: if you look, you will eventually find them. And if you don't look, you won't find them! Their utility will be decided on a per domain basis.

As mentioned in the Introduction, the future of software design and synthesis is in automation. Understanding fundamentals of Computational Design will tell us how to think about program design and synthesis in a structured and principled manner. It is clear that many ideas are being reinvented in different contexts. This is not accidental: it is evidence that we are working toward a general paradigm that we are only now beginning to recognize. Modern mathematics provides a simple language and concepts to express Computational Design and exposes previously unnoticed relationships that can be exploited for pragmatic benefit. This is a step in the right direction.

Acknowledgements. We gratefully acknowledge the helpful comments of K. Czarnecki, P. Kim, Z. Diskin, S. Apel, J. Gray, O. Diaz, and S. Trujillo on earlier drafts of this paper. This work was supported by NSF's Science of Design Project #CCF-0438786 and #CCF-0724979, the Portuguese Science Foundation (FCT) under grant SFRH/BSAB/782/2008, and the Basque Government under the Researchers Training Program.

References

[1] Agrawal, A., Karsai, G., Ledeczi, A.: An End-to-End Domain-Driven Software Development Framework. In: OOPSLA 2003 (2003)

[2] Ancona, D., Damiani, F., Drossopoulou, S.: Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In: POPL 2005 (2005)

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

[4] Antkiewicz, M., Czarnecki, K.: Design Space of Heterogeneous Synchronization. In: Proc. Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE) (2007)

[5] Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE TSE (April 2008)

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

[7] Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. ACM TOSEM 11(2) (April 2002)

[8] Batory, D., Liu, J., Sarvela, J.N.: Refinements and Multi-Dimensional Separation of Concerns. In: ACM SIGSOFT 2003 (2003)

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

[10] Batory, D.: Multi-Level Models in Model Driven Development, Product-Lines, and Metaprogramming. IBM Systems Journal 45(3) (2006)

[11] Batory, D.: Program Refactorings, Program Synthesis, and Model-Driven Design. In: ETAPS 2007, keynote (2007)

[12] Batory, D.: Using Modern Mathematics as an FOSD Modeling Language. In: GPCE 2008 (2008)

[13] Benavides, D., Trinidad, P., Ruiz-Cortes, A.: Automated Reasoning on Feature Models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)

[14] Bézivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: Best Practices for Model-Driven-Software Development (2004)

[15] Bézivin, J., Bouzitouna, S., Del Fabro, M., Gervais, M.-P., Jouault, F., Kolovos, D., Kurtev, I., Paige, R.: A Canonical Scheme for Model Composition. In: ECMDA-FA 2006 (2006)

[16] Cuadrado, J.S., Molina, J.G., Tortosa, M.: RubyTL: A Practical, Extensible Transformation Language. In: ECMDA-FA 2006 (2006)

[17] Freeman, G., Batory, D., Lavender, G.: Lifting Transformational Models of Product Lines: A Case Study. In: ICMT 2008 (2008)

[18] Gray, J.: Private correspondence (July 2008)

[19] Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)

[20] Jackson, D.: Alloy: A Lightweight Object Modeling Notation. In: ACM TOSEM (April 2002)

[21] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Model Transformations in Practice Workshop at MODELS 2005 (2005)

[22] Kiczales, G., et al.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)

[23] Kim, C.H.P., Kaestner, C., Batory, D.: On the Modularity of Feature Interactions. In: GPCE 2008 (2008)

[24] Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-Based DSL Frameworks. In: OOPSLA 2006 (2006)

[25] Lawvere, F.W., Schanuel, S.H.: Conceptual Mathematics: A First Introduction To Categories. Cambridge University Press, Cambridge (1997)

[26] Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: ASE 2001 (2001)

[27] Neema, S.K.: System-Level Synthesis of Adaptive Computing Systems. Ph.D. Vanderbilt University (2001)

[28] Odersky, M., et al.: An Overview of the Scala Programming Language (September 2004), `http://scala.epfl.ch`

[29] Oldevik, J.: UMT: UML Model Transformation Tool Overview and User Guide Documentation (2004), http://umt-qvt.sourceforge.net/docs/

[30] Pierce, B.: Basic Category Theory for Computer Scientists. MIT Press, Cambridge (1991)

[31] Püschel, M., et al.: SPIRAL: Code Generation for DSP Transforms. Proc. IEEE 93#2 (2005); Special Issue on Program Generation, Optimization, and Adaptation

[32] Selinger, P., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access Path Selection in a Relational Database System. In: ACM SIGMOD 1979 (1979)

[33] Sprinkle, J., Karsai, G.: A Domain-Specific Visual Language for Domain Model Evolution. J. Vis. Lang. Comput. 15(3-4) (2004)

[34] Trujillo, S., Azanza, M., Diaz, O.: Generative Metaprogramming. In: GPCE 2007 (2007)

[35] Trujillo, S., Batory, D., Diaz, O.: Feature Oriented Model Driven Development: A Case Study for Portlets. In: ICSE 2007 (2007)
[36] Uzuncaova, E., Garcia, D., Khurshid, S., Batory, D.: Testing Software Product Lines Using Incremental Test Generation. In: ISSRE 2008 (2008)
[37] Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: UniTI: A Unified Transformation Infrastructure. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 31–45. Springer, Heidelberg (2007)
[38] Wachsmuth, G.: Metamodel Adaptation and Model Co-Adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
[39] Wing, J.: Computational Thinking. In: CACM 2006 (March 2006)
[40] Zhang, C., Gao, G., Jacobsen, H.-A.: Towards Just-in-time Middleware Architectures. In: AOSD 2005 (2005)