Generative Programming and Component Engineering (GPCE) 2008

Using Modern Mathematics as an FOSD Modeling Language

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu

Abstract

Modeling languages are a fundamental part of automated software development. MDD, for example, uses UML class diagrams and state machines as languages to define applications. In this paper, we explore how *Feature Oriented Software Development (FOSD)* uses modern mathematics as a modeling language to express the design and synthesis of programs in software product lines, but demands little mathematical sophistication from its users. Doing so has three practical benefits: (1) it offers a simple and principled mathematical description of how FOSD transforms, derives, and relates program artifacts, (2) it exposes previously unrecognized commuting relationships among tool chains, thereby providing new ways to debug tools, and (3) it reveals new ways to optimize software synthesis.

Categories and Subject Descriptors D.2.11 Software Architectures: Languages (e.g., description, interconnection, definition) .

General Terms Design, Theory

Keywords: Commuting Diagrams, Features, Geodesics, Model Driven Design, Software Product Lines.

1. Introduction

Modeling languages are a fundamental part of automated software development. *Model driven design (MDD)*, for example, uses UML class diagrams and state machines as languages to define and synthesize applications [38]. *Service oriented architectures (SOA)* use message sequence charts to specify SOA applications [15].

Feature oriented software development (FOSD) is a compositional paradigm for program synthesis [9]. *Features* are modular increments in program development, and different compositions of features yield different programs. Given a set of features, there can be an exponential number of meaningful compositions, and as such, FOSD is ideally suited for *software product lines (SPLs)*, where a SPL is a family of similar programs that are differentiated by their features.

Although there are many SPL methodologies (e.g. [28][50][59]), FOSD is distinguished in its use of elementary mathematics as a modeling language to express program designs: features are unary functions that transform simple programs to more elaborate programs. A program's design is thus a composition of functions. A further distinction is that FOSD started from practice (i.e., by building product lines via feature composition) and then gradually a mathematics was developed to explain it. This paper is a next step in this practice-towards-theory approach.

AHEAD is an implementation of FOSD. Experience has shown AHEAD concepts and tools are easy to learn. Further, AHEAD has been used to build product lines in a wide range of domains including fire support systems [8], portlets [56], network protocols [7], peer-to-peer communications protocols [4], and the AHEAD tools themselves [1], where the AHEAD tool suite currently exceeds 250K Java LOC.

Recently, we realized that FOSD could benefit from elementary ideas from *Category Theory (CT)*, which is a theory of mathematical structures and their relationships. CT is very abstract, and for typical software developers inaccessible. Yet, its basic ideas have proven quite useful as a modeling language for FOSD [35][56][57]. However, unlike prior work that stresses the formality of CT, our use of CT demands little mathematical sophistication, and certainly makes no contribution to CT or algebraic techniques whatsoever.

But connecting FOSD to CT has at least three important benefits: (1) it offers a simple and principled mathematical description of how FOSD transforms, derives, and relates program artifacts. We see this as a precursor to more formal approaches to software development. (2) It exposes previously unrecognized commuting relationships among tool chains, thereby providing new ways to debug tools. And (3) it reveals new ways to optimize the synthesis of programs. The contribution of this paper is to document each of these benefits. We begin by explaining the core ideas of FOSD.

2. Early FOSD Models of Product Lines

A *feature* is an increment in program development. A *software product line (SPL)* is a family of programs where no two programs have the same combination of features. Every program has multiple representations (e.g., source, documentation, etc.) and adding a feature to a program may elaborate each of its representations. We briefly review the mathematics of the first two generations of FOSD — GenVoca [7] and AHEAD [10] — in this section. A third generation that uses ideas from CT is presented in Section 4.

2.1 GenVoca

GenVoca is a compositional paradigm for defining product lines. Base programs are 0-ary functions called *values*:

GPCE'08, October 19–23, 2008, Nashville, Tennessee, USA.

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

and features are unary functions that elaborate (extend) programs:

```
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 a named expression, e.g.:

```
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 programs that can be created from a set of values and functions is a *product line*. Expression optimization is program design optimization, and expression evaluation is program synthesis [10].

Not all combinations of features are meaningful. The use of some features precludes or demands the use of others. Feature diagrams define the legal combinations of features [20][34]. They are and-or trees, where leaves are primitive features and nonterminals are compound features. In effect, they express a product line as a directed graph where base programs are source nodes (no incoming arrows); the remaining nodes are derived programs. An arrow $D{:}P_1 \rightarrow P_3$ denotes the application of a feature D to program P_1 that produces program P_3. Figure 1 depicts a small FOSD product line.

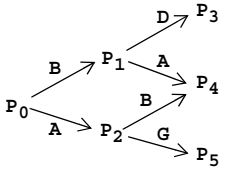

Figure 1. A Product Line

> **Note**: Program P_4 in Figure 1 can be produced in two different ways: start with program P_0 and add features A and B in any order. Features A and B are said to be *commutative*. Commuting features are common in SPLs, although not all pairs of features commute. Also, as a rule, there are no cycles in product line graphs.

2.2 AHEAD

AHEAD generalizes GenVoca in two ways. First it reveals the internal structure of GenVoca values as tuples. Every program has multiple representations, such as source, documentation, bytecode, and makefiles. A GenVoca value is a tuple of program representations. In a product line of parsers, for example, a base parser f is defined by its grammar g_f, Java source s_f, and documentation d_f. Program f has the tuple $f=[g_f, s_f, d_f]$. Each program representation has subrepresentations, and they too have subrepresentations, recursively. In general, a GenVoca value is a tuple of nested tuples that define a hierarchy of representations for a particular program.

> **Example**. Suppose terminal representations are files. In AHEAD, grammar g_f corresponds to a single BNF file, source s_f corresponds to a tuple of Java files $[c_1 \ldots c_n]$, and documentation d_f is a tuple of HTML files $[h_1 \ldots h_k]$. GenVoca values (nested tuples) can be depicted as directed graphs: the graph for program f is Figure 2. Arrows denote projections, i.e., mappings from a tuple to one of its components. AHEAD implements tuples as file directories, so

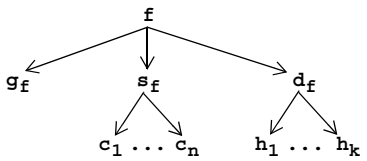

Figure 2. Hierarchical Program Representations

> f is a directory containing file g_f and subdirectories s_f and d_f. Similarly, directory s_f contains files $c_1 \ldots c_n$, and directory d_f contains files $h_1 \ldots h_k$.[1]

Second, AHEAD expresses features as nested tuples of unary functions called *deltas*. Deltas can be program refinements (semantics-preserving transformations), extensions (semantics-extending transformations), or interactions (semantics-altering transformations). We use the neutral term "delta" to represent all of these possibilities, as each occurs in FOSD.

As an example, suppose feature j modifies a grammar by Δg_j (new rules and tokens are added), modifies source code by Δs_j (new classes and members are added and existing methods are modified), and modifies documentation by Δd_j. The tuple of deltas for feature j is $j=[\Delta g_j, \Delta s_j, \Delta d_j]$, which we call a *delta tuple*. Elements of delta tuples can themselves be delta tuples. For example, Δs_j represents the changes that are made to each class in s_f by feature j, i.e., $\Delta s_j = [\Delta c_1 \ldots \Delta c_n]$.[2]

The representations of a program are computed recursively by composing tuples element-wise. The representations for parser p whose GenVoca expression is j•f are:

```
p1 = j•f                              ; GenVoca expression
   = [Δgj,Δsj,Δdj]•[gf,sf,df]         ; substitution
   = [Δgj•gf,Δsj•sf,Δdj•df]           ; compose element-wise
```

That is, the grammar of p is the base grammar composed with its extension ($\Delta g_j \bullet g_f$), the source of p is the base source composed with its extension ($\Delta s_j \bullet s_f$), and so on. As elements of delta tuples can themselves be delta tuples, composition recurses, e.g., $\Delta s_j \bullet s_f = [\Delta c_1 \ldots \Delta c_n] \bullet [c_1 \ldots c_n] = [\Delta c_1 \bullet c_1 \ldots \Delta c_n \bullet c_n]$.

Summarizing, GenVoca values are nested tuples of program artifacts, and features are nested delta tuples, where • recursively composes them. This is the essence of AHEAD [7]. These are the ideas that were used to synthesize programs in many SPLs [7][8][12] [43][56]. Figure 3 shows how the representation or directory hierarchy of program f maps to an isomorphic hierarchy for program p=j•f. Each node of f maps to a corresponding node in p with the same name (although clearly the contents of a node in f may be different than its node in p). Not shown is that each arrow in f maps to the corresponding arrow in p.

---

1. Files can be hierarchically decomposed further. Each Java class can be decomposed into a tuple of members and other class declarations (e.g., initialization blocks, etc.) [10].
2. The value of a tuple component may be 0 (empty) if the corresponding file is undefined. A feature that first defines a file (say x) uses a delta that maps 0 to x.

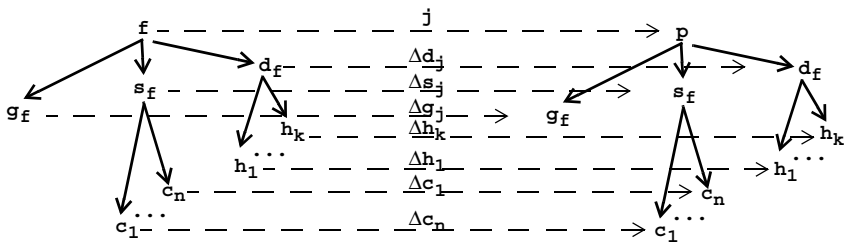

Figure 3. Features Map Representation Hierarchies

```
class foo {
   int x = 0;
   void inc() {
     x++;
   }
}
```
(a) **W**

```
refines class foo {
   int y;
   void set() { y=x; }
   void inc() {
     y++; super.inc();
   }
}
```
(b) **R**

```
class foo {
   int x = 0;
   int y;
   set() { y=x; }
   void inc() {
     y++; x++;
   }
}
```
(c) **R•W**

```
"+" PLUS

Expr : Val
     | Val Opr Expr ;
Val  : INTEGER ;
Opr  : PLUS ;
```
(d) **G**

```
"-" MINUS

Opr : super
    | MINUS ;
```
(e) **T**

```
"+" PLUS
"-" MINUS

Expr : Val
     | Val Opr Expr ;
Val  : INTEGER ;
Opr  : PLUS
     | MINUS ;
```
(f) **T•G**

Figure 4. Values, Functions, and Their Composition

2.3 Feature Implementations

There are many ways to implement AHEAD. Some of AHEAD's basic ideas are now found in contemporary languages that support collaboration-based designs [4][24][47]. We briefly reviews its basics; for more details see [1][10].

The unary functions that can be defined in AHEAD are simple: new elements can be added to a file and existing elements can be altered. Figure 4a shows a value `W` which represents Java class `foo`. A delta (unary function) of `W` is `R`, shown in Figure 4b. `R` means "add field `int y`, method `void set()` to class `foo`, and extend method `inc()`". Method deltas are written and interpreted just like method overrides in Java subclassing hierarchies [1]. The composition of `R•W` is the class `foo` of Figure 4c. The same ideas apply to other (non-Java) program representations. For example, Figure 4d is a base grammar `G`, Figure 4e is a delta `T` that adds a token `MINUS` and a new right-hand side to production `Opr`. The "`super`" construct refers to the prior right-hand sides of a production (in this case, `Opr`). The composition `T•G` is the grammar of Figure 4f.

The benefit of using similar delta concepts for different program representations is pragmatic. If each representation had a completely different mental model for deltas, the ability of any individual to understand all of them and use them effectively rapidly diminishes. Our experience is that uniformity contributes to understandability and simplicity. Note: FOSD *does not* preclude other and more sophisticated ways of defining deltas. Aspects and rule sets of transformation systems are examples [13][36]. Both technologies could be (and have been!) uniformly applied to all kinds of program representations [30].

3. Modeling GenVoca and AHEAD using CT

3.1 Categories

A *category* is a directed multigraph with special structure. Nodes are *objects* and edges are *arrows*. An arrow drawn from object `X` to object `Y` is a map with `X` as its domain and `Y` as its codomain. Arrows compose and arrow composition is associative. Also, there are identity maps for each object, indicated by loops [49]. See Figure 5.

A product line is a category: Figure 1 is identical to Figure 5, minus identity arrows. (Identity and composed arrows are henceforth omitted for readability). Each object P_i in Figure 5 is a domain with one element — the *i*th program in the product line. Let `P` denote the category of an SPL, such as Figure 5.[3]

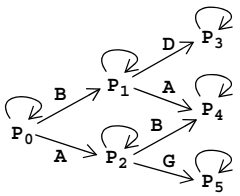

Figure 5. A Category

Arrows of a category are unary (single-parameter) maps. To express maps with multiple inputs or outputs, tuples of objects are used. A tuple is formed by a *product of objects*. So if category `C` had objects O_1 and O_2, a product of these objects, $[O_1,O_2]$, becomes another object of `C`. In general, a *product of a family of objects* $O_1 \ldots O_n$ is a tuple $[O_1 \ldots O_n]$. A projection arrow/function is used to obtain a particular object of a tuple.

3. Just as partial orders and containment lattices are representative of elementary or trivial categories, so too are SPLs.

To see products in action, the nested tuples of AHEAD values define a category. Figure 2 depicts a category minus identity and composed arrows. The leaves are objects representing domains with a single file. Each non-leaf is a product of a family of objects, e.g., s_f is the product of objects $c_1 \ldots c_n$ and f is the product of g_f, s_f, and d_f [49]. Arrows are element projection functions.

The relationship between P, the category of all programs in an SPL (Figure 1), and R, the category of all program representations (Figure 2), is expressed by the categorical product ($R \times P$), where an arrow represents either a delta or a projection function. Although categorical products are not essential to our main-line discussion, they are relevant and are explained in Appendix I.

3.2 Functors

A *functor* is a structure preserving map between two categories [49]: functor $F: A \rightarrow B$ maps each object in A to an object in B, and each arrow in A to an arrow in B such that the connectivities of A is preserved. The functors that arise in AHEAD are particularly simple: they are maps between isomorphic categories, such as maps from one hierarchy of representations to another (Figure 3), and as we will show later one product line to another (Figure 15).

> **Example**. A feature $F: A \rightarrow B$ is a functor that maps the category of representations of program A to the category of representations of program B. In general, a GenVoca expression can be seen as the application of a series of functors to an initial category (the category of representations of a base program). Figure 3 is an example: f is the initial category and j is a functor applied to f. Each object in f is mapped to the corresponding object in $p = f \bullet j$ and each arrow in f is mapped to the corresponding arrow in p.

Relating features to functors offers an interesting perspective on today's industrial programming languages (e.g., Java and C#) and contemporary programming language research. Industrial languages enable engineers to define objects (program source), but offer little or no help in defining and composing arrows (e.g., deltas). That is, common languages do not provide AHEAD-like `refines` declarations to define deltas that update existing class and method declarations and that allow such deltas to be composed. In absence of language support, programmers resort to preprocessors or transformation systems to implement deltas, which have important drawbacks. Namely, the language to express deltas is different from the source language, deltas cannot be compiled separately, creating and maintaining the infrastructure to define and compose deltas may burden programmers, etc.

From a CT perspective, half of a fundamental picture is missing: modeling languages and programming languages should allow developers to define not only objects (programs), but arrows (program deltas), and arrow compositions as well. Recent programming languages that support collaboration-based designs can be recognized as attempts to address this omission, e.g., virtual classes and mixins [14][45], Scala [47], higher-order hierarchies [24], and giving aspects functional semantics [44].

4. Next Generation FOSD Model of Product Lines

AHEAD captures the lock-step update of artifacts (files, models) of a program when features are applied. It does not capture important derivation relationships among artifacts. The next generation of FOSD integrates ideas of MDD that emphasizes the derivation of one model (artifact) from another. This next-generation of FOSD is called *Feature Oriented MDD (FOMDD)* [56][57].

Suppose we generalize the tuples of our parser product line. In addition to a grammar (g), source code (s), and documentation (d) representations, we add a bytecode (b) representation. So base parser $f = [g_f, s_f, d_f, b_f]$ now has four components, one for each representation. Derivation relationships exist among components of f: bytecode (b) is derived from source (s) by the Java compiler. That is, `javac` is a function that maps Java source to Java bytecodes. Similarly, the documentation (d) may also be derived from the Java source (s). For example, `javadoc` is a function that maps Java source to HTML documentation. As a general rule, common tools used by software engineers implement object-to-object maps in FOSD.

Note that there are other basic CT concepts that arise in FOSD, such as natural transformations. As these ideas are not essential to our main-line discussion, they are presented in Appendix II.

4.1 Commuting Diagrams

The commuting diagram [39] [49] of Figure 6 expresses what we expect to hold between derivations and deltas. Horizontal arrows are deltas (called *endogenous transformations* in the MDD literature [46]) and vertical arrows are derivations (called *exogenous transformations* [46]). Any path from the upper-left object to the lower-right object produces an equivalent result. In Figure 6, a higher-order function or *operator* Θ maps delta ΔM to delta ΔN. The general relationship is:

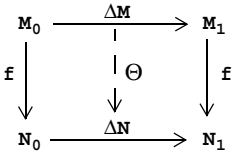

Figure 6. Commuting Diagram

$$f \bullet \Delta M = \Theta(\Delta M) \bullet f$$

As CT is not constructive in the sense that it doesn't tell you how to implement arrows, it can be a substantial engineering challenge to create operator Θ. As an example, Figure 7 depicts how `javac` relates the Java source of programs P_0 and P_1 (namely S_0 and S_1) to their corresponding bytecodes B_0 and B_1. The horizontal arrow $S_0 \rightarrow S_1$ is a source code delta ΔS (i.e., a set of AHEAD class additions and class deltas), and the $B_0 \rightarrow B_1$ arrow is the corresponding bytecode delta ΔB. We implemented a special tool in AHEAD (`bccompiler`) to implement the arrow-to-arrow mappings of Figure 7 (i.e., ΔB = `bccompiler`(ΔS)) [3]. That is, `bccompiler` allowed us to separately compile the files of ΔS into bytecode (ΔB). This allowed us to demonstrate that extending

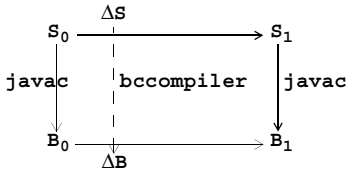

Figure 7. Source-Bytecode Diagram

source code s_0 by Δs and compiling was equivalent to compiling the source of s_0 and extending its bytecode by Δb.

Commuting diagrams such as Figure 7 have practical uses. One is immediately evident: most AHEAD tools are preprocessors that map source files to extended source files. If features are to be distributed commercially as components, bytecode (not source code) will be the preferred representation.

> **Note**. Our `bccompiler` operator can add new classes, new members to existing classes, and can wrap existing methods. However, `bccompiler` relied on `javac`, which propagates constants, and this can be problematic when different features assign different values to variables. So although we demonstrated the feasibility of bytecode deltas, a more general approach, called *separate class compilation*, is needed, which delays the folding of constants and other optimizations until bytecode composition time [1][2].

Proofs should accompany commuting diagrams, but the scale of programs in AHEAD puts this beyond the state of the art in program verification. For example, the Sun Java 1.6.1 compiler maps Java programs to bytecode, but we are unaware of a correctness proof. Similarly, we do not have proofs that the semantic properties of features are preserved *or correctly transformed* by derivations and deltas. Proving properties of arrows on the scale of Figure 7 is appropriate for the Verified Software Grand Challenge of Hoare, Misra, and Shankar [40], which seeks scalable technologies for program verification.

We use commuting diagrams to define relationships that we expect to hold among program artifacts. It is this 'engineering' or 'informal' approach to CT, rather than a rigorous mathematical approach, that we have found useful. In the absence of proofs, we take a standard software engineering line: forms of equivalence are demonstrated by testing. That is, we start with program $s_0 \in S_0$ and produce programs $b_1 = \text{javac}(\Delta S \bullet s_0)$ and $b_1' = \text{bccompiler}(\Delta S) \bullet \text{javac}(s_0)$ and test their equivalence. In the case of bytecodes, b_1 and b_1' are subjected to the same system or integration tests; if both have the same responses, these programs are considered behaviorally identical for those tests. In the case that a commuting diagram yields a source document, "diffs" can be used to test for source equivalence. (*Source equivalence* is syntactic equivalence with two relaxations: it allows permutations of members when member ordering is not significant and it allows white space to differ when white space is unimportant).

Experience to date is that FOMDD exposes commuting relationships in SPL models, and imposes a similar number of constraints on the commutativity of *tool chains*, i.e., chains of composed tools or transformations. We were unaware of many of these constraints. Not surprisingly, our tools initially failed to satisfy commuting relationships. By repairing our tools so that they did, we have greater confidence in our tools and in our understanding of our domain. Both are wins from an engineering perspective: we can reason algebraically about our designs, rather than hacking code. This is a good example where FOMDD exposes valuable and previously unrecognized properties that program synthesis tools and tool chains must satisfy. In the next section, we see an interesting twist on the use of commuting diagrams.

4.2 Geodesics

Suppose program P_0 has a specification from which multiple representations are derived. Figure 8a shows P_0 consisting of three representations that are derived from the topmost object. (In our tuple notation, $P_0 = [r_0, f \bullet r_0, g \bullet f \bullet r_0]$ where `f` and `g`, are unary functions and r_0 is a base program representation). As features are composed onto P_0, a mesh of commuting diagrams is produced. The geometry of Figure 8b is regular, although it could just as easily be ragged (Figure 8c). Meshes are created by translating delta arrows that connect the topmost objects into delta arrows of lower-rung objects. Ragged geometries arise when delta arrows are not implemented.

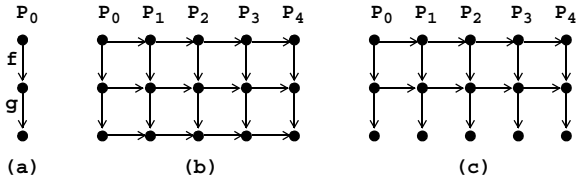

Figure 8. Geometries

> **Note**: In principle, absent arrows always exist, but there is no tool to compute them. For example, until we built the `bccompiler` tool, we could not materialize the $B_0 \rightarrow B_1$ arrow of Figure 7. As mentioned earlier, building tools that implement arrows can be a significant engineering challenge.

Given an object in the upper left corner of a diagram, we want to compute the object in the lower right. Any path will produce the desired result. For a rectangular mesh of $m \times n$ nodes, there are $\binom{m+n-2}{m-1}$ such paths.

We have observed that engineers develop programs by creating a single path (tool chain) that maps the initial object (model) to the target object (model) in a commuting diagram. Such a path is called a *makefile*. They (including us) were not aware of other possible paths. CT exposes new ways to synthesize programs, and this has lead us to interesting results.

From an engineering perspective, creating and/or traversing an arrow has a cost, and not all arrows have the same cost. Given a metric that defines the cost of traversing (synthesizing and composing) an arrow, diagram geometries warp (Figure 9). No longer are all paths equidistant. It becomes an optimization problem to determine the shortest traversal, called a *geodesic,* to synthesize a target result. The next sections sketch the utility of geodesics and the generalization of geodesics to more complex categories.

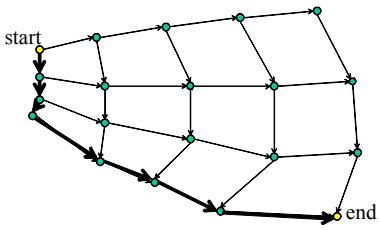

Figure 9. Geodesic

4.3 Applications of Geodesics

4.3.1 PinkCreek

PinkCreek is a product line of web portlets (i.e., web components) and was the first FOMDD application [56]. A category of objects (a.k.a. *models*) is displayed in Figure 10a, where a series of different portlet representations were derived from the topmost object. As features are composed, a multi-pleated geometry is created (Figure 10b).

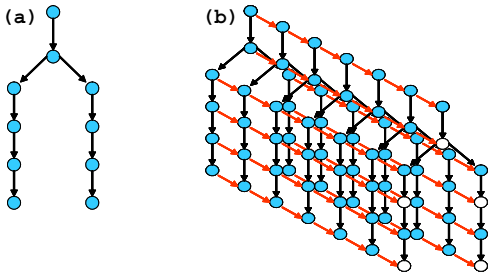

Figure 10. PinkCreek Geometries

A PinkCreek geodesic is not a line. Starting at the upper left object (which corresponds to a base statechart), there is a set of target objects produced by the final feature that are to be computed (indicated by white objects in Figure 10b). A common approach in MDD is to traverse the ridge of the geometry and follow the arrows downward. "Traversing the ridge" means extend the original statechart (the top-most object of Figure 10a) into its final form, and then derive its representations. FOMDD predicts the existence of other paths, which might be more efficient. Such a path was discovered experimentally: start from the original statechart, derive its representations, and then apply features to elaborate the desired representations. This new approach was 2-3 times faster than the original traversal and was a consequence of special optimizations that were possible in the PinkCreek design [56].

When there is only one source object and one target object, a geodesic can be computed by Dijkstra's shortest path algorithm. In PinkCreek, there is one starting object and *n* target objects and computing a geodesic requires solving the Directed Steiner Tree Problem, which is NP-hard [19]. More generally, a geodesic can have *m* starting objects and *n* target objects. While simple heuristics for computing geodesics may suffice, there may be some interesting optimization problems to be addressed.

4.3.2 Testing Software Product Lines

Testing SPLs is a fundamental problem. Not only should it be possible to generate any program of a product line, it should also be possible to generate tests for that program to provide evidence that the generated program is correct.

Specification-based testing can be an effective approach for testing the correctness of programs [17][18][32]. The idea is to map a program's specification automatically to a set of test inputs. These inputs are fed to the program, and the program's response can be validated automatically using correctness criteria. Alloy is an example [32][33]. An Alloy specification S for program P defines properties (constraints) that data structures must satisfy. The Alloy analyzer [33] translates S to test inputs T in the following way: the analyzer converts S into a propositional formula, the formula is solved by a SAT solver, and each solution is converted into a test.

In the context of SPLs, each program P_i is represented as an ordered pair $[S_i, T_i]$, where S_i is a specification of P_i and T_i is its set of tests. The commuting diagram of Figure 11 shows a product line of four programs (P_0-P_3), where horizontal arrows are deltas and vertical arrows are the mappings of Alloy.

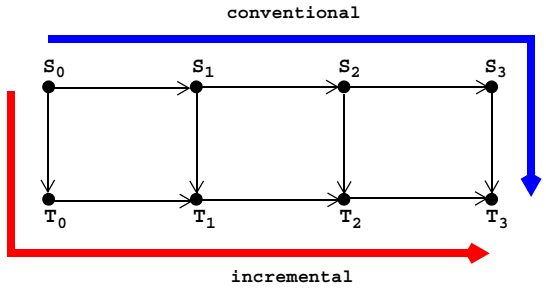

Figure 11. Conventional vs. Incremental Test Generation

In a conventional approach, one starts with base program P_0 which has Alloy spec S_0. Suppose feature F (which maps P_0 to P_1) has Alloy specification S_F. When F is composed with P_0 to produce program P_1 (i.e., $P_1 = F \bullet P_0$), assume that the composite specification is $S_1 = S_F \wedge S_0$ (i.e., the conjunction of the F and P_0 specs). Adding two more features produces spec S_3, at which point the Alloy analyzer translates S_3 into T_3.

FOMDD predicts Figure 11 that reveals other ways of producing test T_3 starting from specification S_0. *The challenge is that it is not obvious how to take any path other than the conventional path — no other path has ever been taken*.

A way to traverse other paths was proposed by Uzuncaova, et al. [60][61]. Instead of solving the entire formula S_3 (as is done conventionally), an alternative is to find a solution I_0 to the base spec S_0, and then use I_0 *as a constraint* for solving more complex spec S_1. A solution for I_1 is then used as a constraint for solving S_2, and so on. That is, start with the solution (tests) of a simpler program, and extend it to a solution (tests) for a more complex program. The *incremental approach*, as it is called, has appealing properties. First, it is sound: any solution of S_{i+1} that can be computed from I_i is, obviously, a solution of S_{i+1}. Second and more interesting, it is complete: any solution to S_{i+1} must embed a solution to subproblem S_i. Thus, by iterating over solutions to S_i, it is possible to enumerate *all* solutions of S_{i+1}.

> **Note**: some solutions to S_i may not extend to solutions of S_{i+1}, and some S_i solutions may extend to multiple S_{i+1} solutions.

> **Note**: the geometry of this problem matches that of Figure 8c: the top nodes are Alloy specifications S_i, the middle nodes are solutions I_i, and the bottom nodes are tests T_i, where solutions are refined, and tests are derived from these solutions.

Initial experimental results comparing the incremental approach with the conventional approach were encouraging. Figure 12

subject	product	speed-up
Binary Tree (scope=10)	base	n/a
	size•base	0.58×
	parent•base	0.72×
	search•base	27.99×
	parent•size•base	0.41×
	search•size•base	55.16×
	search•parent•base	21.40×
	seaarch•parent•size•base	66.44×
INS (scope=16)	base	n/a
	attr-val•base	0.35×
	label•attr-val•base	14.53×
	record•label•attr-val•base	9.56×

Figure 12. Conventional v.s Incremental Test Generation

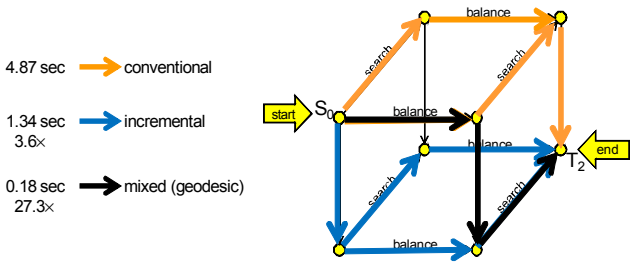

Figure 13. Geodesic in a Commuting Diagram

shows the time for creating tests for a product line of trees (a standard example of researchers using the Alloy analyzer). For some experiments, the conventional approach was faster. The reason is that the composite predicate was simple enough to solve directly — it was overkill to partition it into smaller predicates, solve these predicates, and then extend their solutions. However, for a majority of cases, the conventional approach to solve a composite predicate directly was often more than an order of magnitude *slower* than an incremental approach. In several cases, an incremental approach was 50× faster. The reason is that it is easier to find solutions to simple predicates and to extend their solutions.

Another interesting result is that it is possible to permute the order in which features are composed. Although the technical details for how this is can be done for arbitrary program artifacts is beyond the scope of this paper (see [5][37]), in principle, the idea is clear for the way Alloy specifications are composed. Figure 13 shows the construction of tests for a balanced search tree; the different ways in which a tree specification ($\mathtt{S_0}$) can be mapped to the tests for a balanced search tree ($\mathtt{T_2}$) can be visualized by a 3-dimensional commuting diagram. Note that the conventional and incremental approaches correspond to some paths through this diagram. We evaluated all paths.

Conventional paths traverse the top of the cube starting at $\mathtt{S_0}$ and lastly deriving the test $\mathtt{T_2}$ from the full specification of $\mathtt{S_2}$. The fastest this could be accomplished was in 4.87 seconds. Incremental paths derived test $\mathtt{T_0}$ immediately, and traversed the bottom of the cube to $\mathtt{T_2}$. The fastest that this could be accomplished was in 1.34 seconds, a factor of 3.6× improvement. However, neither of these traversals was a geodesic: the fastest traversal is formed by first refining $\mathtt{S_0}$ by the **`balance`** feature, then deriving the test for balanced trees, and finally extending this test by the **`search`** feature to $\mathtt{T_2}$. This path was traversed in .18 seconds, a 27.3× factor improvement over the conventional approach. Once again, it is an interesting and on-going research problem to determine heuristics for identifying optimal paths (geodesics) for test generation.

5. Related Work

There is a huge literature at the intersection of category theory and *Computer Science (CS)*. Papers appearing in software engineering venues are often theoretical, requiring mathematical expertise to appreciate their contributions. Even good tutorials, like [53] which illustrates CT concepts using the ML programming language, are difficult to relate to practical design problems encountered in extensible object-oriented programs and feature-based SPLs. (Part of the problem is that it is hard for ML functors to express the incremental addition and refinement of methods and variables to class-like structures). We know that useful CT connections have been made with non-trivial applications (e.g., [23][25][48]), but the key challenge is finding connections and examples that can be appreciated by typical SPL designers. It is important to remember that we did not set out to develop FOMDD with CT in mind. On the contrary, FOMDD was created and *later* we discovered its connection with CT. (It is this subsequent connection with CT that this paper documents). Our use of CT as an informal way to model SPL domains and to expose the commuting relationships among program synthesis tools and features is the primary contribution of our work. Other researchers are finding similar benefits of using commuting diagrams informally to structure and explain their systems (e.g., version control [52], feature interaction [5][37]).

Research in algebraic specification uses CT and commuting diagrams to express ideas similar to those in this paper [23][48]. In fact, the basic notion that refinements affect different representations of a module (e.g., its interface, implementation, parameters) is present, although our use of deltas (which need not be semantics-preserving transformations) is a clear difference. Terminology is misaligned: the terms 'refinement' and 'extension' have different meanings. Consider a 2-space, where points along the X-axis are specifications, and points along the Y-axis are implementations. A common paradigm (e.g., Z [55]) builds a specification incrementally by extensions. Once completed, the specification is then refined progressively to an implementation. Program `P` has specification `S` and implementation `I` in Figure 14a, where horizontal arrows denote specification extensions and vertical arrows are refinements.

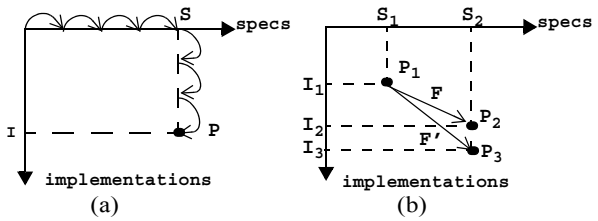

Figure 14. Extension and Refinement

FOSD is different. A feature extends both a specification and an implementation, i.e., features move diagonally through this 2-space. In Figure 14b, program $\mathtt{P_1=[S_1,I_1]}$ is mapped to program

P_2=[S_2,I_2] by feature F. It is possible for features to share the same specification extension (e.g., F′ in Figure 14b) but have a very different implementation (e.g., P_3=[S_2,I_3]), which leads to interesting optimizations. For example, which feature produces the most efficient program [54]? To the best of our knowledge, features implement a form of constrained subtypes [42], where type specifications are very weak.

Specware uses CT as a formal foundation for program synthesis [48]. Specifications are composed by pushouts, which are commutative. Pushouts are appropriate as the specifications that are composed in Specware are orthogonal (i.e., pushouts effectively compute specification union). FOSD uses a different model: feature composition is function composition, which is not commutative. To illustrate, suppose feature F_1 increments field V, and feature F_2 doubles V. The order in which F_1 and F_2 are composed matters. This example leads to inconsistencies in Specware. Of course, a major advantage of Specware are guarantees of correctness in the code that it synthesizes; AHEAD offers no such assurances.

Our work exploits other fundamental ideas in CS. Equating tools with functions or operations is an ancient idea. Among its first statements are the T-diagrams of Early and Sturgis [22] and more recently by Appel [6]. T-diagrams are a graphical way to show the composition of compilers and translators to achieve a particular translator (here called an arrow). An algorithm is given in [22] to show how a desired arrow can (or cannot) be synthesized given a set of primitive arrows. Although unstated, it is an obvious step from here to see how this algorithm can be used to generate multiple ways to synthesize a given arrow, where a "geodesic" would be the cheapest. Our work shows how similar ideas arise in a much more general context, where software design and construction is viewed as a computation. It is this mathematical (transformational) approach that allows us to make connections from software design to elementary ideas in mathematics.

From the FOSD perspective, theorems and proofs are other syntactic artifacts that are subject to transformation. First steps on how theorems (both their statements and proofs) are transformed by features have been taken [12]. Ideally, features define conservative refinements, so that semantic properties that were true before remain true (or are qualified) after a feature has been applied. But in general, there are domains where features have a more invasive impact, where properties may be erased and replaced by others, so that it is not obvious how the replaced definitions can be incrementally built. This abrupt discontinuity in semantics is often called *feature interaction* [16]. The way a pair of features "interact" now is manifested by the need for a third feature to coordinate/modify the activities of the first two features in a rational way [37]. Understanding feature interactions remains a difficult challenge.

FOSD has similar goals to Goguen's parameterized programming [26]. His work offers two distinct forms of parameterization, horizontal and vertical, and uses views to define morphisms (maps) between module interfaces that would otherwise not be composable. Although FOSD emphasizes vertical parameters to express deltas, features (as mentioned in footnote 1) can indeed have horizontal (e.g., performance) parameters.

Collaborations (or role-based designs) were perhaps the first object-oriented way to express features (arrows) [51]. Collaborations can be implemented by virtual classes [45] and mixins [14], and have been the basis of several feature-based design methodologies [58]. Unfortunately, support for collaborations has not found its way into industrial programming languages.

Ernst's *Higher-Order Hierarchies (HOH*) has much of the flavor of AHEAD, where any number of (virtual) classes in an inheritance hierarchy can be extended lock-step. Different extensions can be composed and there are statements to specify compositions [24]. Scala is another language that can express code deltas [47]. Scala is general, and requires programmers to express "type plumbing", i.e., type bindings in deltas. AHEAD's `refines class` construct is much more limited, and hides (or rather assumes) type bindings. Consequently, `refines class` declarations are more compact and may be easier for typical programmers to use [43]. A major difference is Scala has a type system; AHEAD does not.

6. Conclusions

Software architects are not (and may never be) mathematicians, but this should not prevent them from using modern mathematics to express fundamental relationships in program structure, SPLs, and program synthesis. Indeed, conceiving programs as structures (values) and transformations that map programs to other programs is both a fundamental and ancient idea in computer science, although it seems to have been lost in current software design texts and practices.

In this paper, we used elementary concepts of CT as a modeling language to explain how FOSD defines and creates software product lines; we did not use CT as the basis for a formal model for proving theorems about SPL semantics. Rather we used CT informally (a) to explain how in FOSD artifacts that represent programs are transformed to other artifacts by features or by tools, (b) to expose previously unrecognized commuting relationships among tool chains (providing new ways to debug existing tools), and (c) to reveal new ways to optimize program synthesis. All of these advances were beneficial in building practical tools and designing FOSD models of SPLs.

We believe that connecting software design and development to mathematics is a precursor to more formal, structured, and repeatable models of automated software development that could be used in practice. Although the integration of program semantics into FOSD is still in its infancy, we believe our work takes us a small step closer to explain the design activities of today's software engineers in a principled way.

Acknowledgements. I greatly appreciate the comments from P. Kim, D. Smith, M. Mehlich, G. Lavender, M. Poppleton, S. Nedunuri, W. Cook, S. Apel, S. Trujillo, O. Diaz, A. Rauschmayer, M. Wirsing, E. Boerger, and J. Misra on earlier drafts of this paper. Also, I thank C. Lengauer for recognizing the vector description of AHEAD, and V. Ramachandran and R. Chowdhury for their help in connecting my work with the Directed Steiner Tree Problem, and discussions with J. McGregor on product line testing. Finally, I thank the referees for their helpful comments. This work was sup-

ported by NSF's Science of Design Project #CCF-0438786 and #CCF-0724979.

7 References

[1] D. Ancona, G. Lagorio, and E. Zucca. "Jam—Designing a Java Extension with Mixins", *ACM TOPLAS 2003*.

[2] D. Ancona, F. Damiani, and S. Drossopoulou. "Polymorphic Bytecode: Compositional Compilation for Java-like Languages", *POPL 2005*.

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

[4] S. Apel, "The Role of Features and Aspects in Software Development", Ph.D. Dept. of Tech.l and Business Info. Systems, University of Magdeburg, Germany, March 2007.

[5] S. Apel, C. Kaestner, and D. Batory. "Program Refactoring using Functional Aspects". *GPCE 2008*.

[6] A. Appel, "Axiomatic Bootstrapping: A Guide for Compiler Hackers", *ACM TOPLAS* Nov. 1994.

[7] D. Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components". *ACM TOSEM*, October 1992.

[8] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study", *ACM TOSEM*, April 2002.

[9] D. Batory, "A Science of Software Design", *AMAST 2004*.

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

[11] D. Batory, "From Implemention to Theory in Product Synthesis", University of Texas Austin, Dept. Comp. Science TR-07-35, June 2007.

[12] D. Batory and E. Börger. "On The Modularization of Theorems for Software Product Lines", to appear in *JUCS*.

[13] I.D. Baxter. "Design Maintenance Systems". *CACM*, April 1992.

[14] G. Bracha and W. Cook. "Mixin-Based Inheritance". *OOPSLA and ECOOP 1990*.

[15] M. Broy, I. H. Krüger, and M. Meisinger. "A Formal Model of Services". *ACM TOSEM*, vol. 16, no. 1, p. 5, Feb. 2007.

[16] M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec. "Feature Interaction: A critical Review and Considered Forecast". *Computer Networks*, January 2003.

[17] J. Chang and D.J. Richardson. "Structural Specification-Based Testing: Automated Support and Experimental Evaluation". *ACM SIGSOFT/FSE 1999*.

[18] Y. Cheon and G.T. Leavens. "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way". *ECOOP 2002*.

[19] M. Charikar, et al., "Approximation Algorithms for Directed Steiner Tree Problems, *ACM-SIAM Symposium on Discrete Algorithms (SODA) 1998*.

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

[21] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[22] J. Earley and H. Sturgis. "A formalism for translator interactions". *CACM Oct. 1970*.

[23] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, Springer-Verlag, 1990.

[24] E. Ernst, "Higher Order Hierarchies", *ECOOP 2003*.

[25] J. Fiadeiro. *Categories for Software Engineers*. Springer 1998.

[26] J. Goguen. "Principles of Parameterized Programming" in T. Biggerstaff and A. Perlis, *Software Reusability Volume II: Applications and Experiences*, Addison-Wesley, 1990.

[27] J. Goguen. "A Categorical Manifesto". *Mathematical Structures in Computer Science*, 1991.

[28] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison-Wesley 2005.

[29] J. Goodenough and S. Gerhart. "Toward a Theory of Test Data Selection". *IEEE TSE*, June 1975.

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

[31] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. "Generating Finite State Machines From Abstract State Machines". *ISSTA 2002*.

[32] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.

[33] D. Jackson. "Alloy: A Lightweight Object Modeling Notation". *ACM TOSEM*, April 2002.

[34] K. Kang, et al., "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Tech Report CMU/SEI-90-TR-21.

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

[36] G. Kiczales, et al. "An Overview of AspectJ". *ECOOP 2001.*

[37] C.H.P. Kim, C. Kästner, and D. Batory. "On the Modularity of Feature Interactions". *GPCE 2008*.

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

[39] F.W. Lawvere and S.H. Schanuel, *Conceptual Mathematics: A First Introduction To Categories*, Cambridge University Press, 1997.

[40] G. Leavens, et al. "Roadmap for Enhanced Languages and Methods to Aid Verification". *GPCE 2006*.

[41] H.C. Li, S. Krishnamurthi, and K. Fisler. "Modular Verification of Open Features Through Three-Valued Model Checking", *Automated Software Engineering Journal*, 2005.

[42] B. Liskov and J.M. Wing, "A Behavioral Notion of Subtyping", *ACM TOPLAS 1994*.

[43] R. Lopez-Herrejon, D. Batory, and W. Cook. "Evaluating Support for Features in Advanced Modularization Technologies", *ECOOP 2005*.

[44] R. Lopez-Herrejon, D. Batory, and C. Lengauer. "A Disciplined Approach to Aspect Composition", *PEPM 2006*.

[45] O.L. Madsen and B. Møller-Pedersen, "Virtual Classes: A Powerful Mechanism in Object-Oriented Programming", *OOPSLA 1989*.

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

[47] M. Odersky, et al. "An Overview of the Scala Programming Language". September 2004, `scala.epfl.ch`

[48] D. Pavlovic and D.R. Smith. "Software Development by Refinement", UNU/IIST 10th Anniversary Colloquium, Formal Methods at the Crossroads: From Panaea to Foundational Support, Springer-Verlag LNCS 2757, 2003.

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

[50] K. Pohl, G. Boeckle, F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques,* Springer 2005.

[51] T. Reenskaug, et al. "OORASS: Seamless support for the creation and maintenance of object-oriented systems". *Journal of Object-Oriented Programming*, October 1992.

[52] D. Roundy, "Theory of Patches", `//darcs.net/manual/node8.html`

[53] D.E. Rydeheard and R.M. Burstall, *Computational Category Theory*, Prentice Hall, 1988.

[54] P. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. "Access Path Selection in a Relational Database System", *ACM SIGMOD 1979*.

[55] J.M. Spivey, *The Z Notation: A Reference Manual*, Oxford University Press, 1998.

[56] S. Trujillo, D. Batory, and O. Diaz. "Feature Oriented Model Driven Development: A Case Study for Portlets", *ICSE 2006*.

[57] S. Trujillo, M. Azanza, and O. Diaz. "Generative Metaprogramming", *GPCE 2007*.

[58] M. VanHilst and D. Notkin. "Using role components to implement collaboration-based designs". *OOPSLA 1996.*

[59] D.M. Weiss, C.T.R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process,* Addison-Wesley, 1999.

[60] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory, "A Specification-Based Approach to Testing Software Product Lines (Poster Paper)". *ACM SIGSOFT/FSE 2007*.

[61] E. Uzuncaova and S. Khurshid. "Constraint Prioritization for Efficient Analysis of Declarative Models". *Symposium on Formal Methods (FM)*, May 2008.

Appendix I. Product of Categories

The *product of categories* `C` and `D`, denoted `C×D`, is the cross product of graphs `C` and `D`; it is formed by pairing each object in `C` with each object in `D`. Let C_i be an object in `C`, and D_j be an object in `D`. An arrow from $[C_i,D_j]$ to $[C_k,D_l]$ is a pairing of two arrows, one $C_i \rightarrow C_k$ in category `C` and another $D_j \rightarrow D_l$ in `D` [49]. Figure 15 illustrates the idea (sans identity and composed arrows).

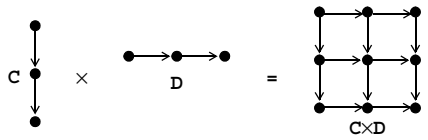

Figure 15. Product of Categories

Recall that a program in AHEAD is a hierarchy of representations. When a feature is applied to a program, the contents of the program's hierarchy is updated. Let `P` be a category of a product line and `R` be a (hierarchical) category of program representations. The product `R×P` defines the relationships between the representation hierarchies of programs in a product line. Figure 3 shows a small part of `R×P` where the representation hierarchies of programs `f` and `j•f` are related via arrows. (Actually, Figure 3 shows the full product `R×P` if `P` contains only programs `f` and `j•f` and arrow `j`).

Appendix II. Natural Transformations

Informally, a *natural transformation (NT)* is a mapping from an object to an arrow [49].[4] NTs arise when derivation relationships are exposed. Consider Figure 16. Suppose `P` (the top category) represents a product line of parsers, where each object P_i is an ordered pair $[S_i,D_i]$ of the program's source code and documentation. Different projections of `P` yield categories `S` and `D`. Category `S` is the product line of the source representations of parsers, and category `D` is the product line of the `javadoc` representations of these parsers. Let `S+D` denote the coproduct (disjoint union) of `S` and `D` [49]. The projection of `P` to `S` is the functor `P2S:P→S+D` and the projection of `P` to `D` is the functor `P2D:P→S+D`. The arrow (tool) `javadoc` defines the object-to-object maps of the natural transformation from `P2S` to `P2D`.

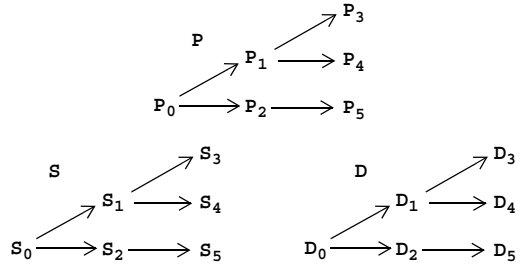

Figure 16. Categories of Program Representations

As mentioned earlier, the functors that arise in AHEAD are maps between isomorphic categories; the functor that maps product line `S` to product line `D` in Figure 16 is example. Other concepts (e.g., limits, equalizers) can be similarly illustrated.

4. Stated differently, a functor `F:A→B` is the embedding of the image of category `A` in the image of category `B`. A natural transformation is primarily a map from one embedding to another, and secondarily as a map from objects in the source category to arrows between the images under the functors of that object in the target category.