A Modeling Language for Program Design and Synthesis

Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract. Software engineers define structures called programs and use tools to manipulate, transform, and analyze them. A modeling language is needed to express program design and synthesis as a computation, and elementary algebra fits the bill. I review recent results in automated software design, testing, and maintenance and use the language of elementary mathematics to explain and relate them. Doing so outlines a general and simple way to express and understand the relationships between different topics in program synthesis.

1 Introduction

A goal of software engineering research is to understand better how programs can be developed automatically for well-understood domains. We know the problems of building such programs, we know the solutions, but all too often these programs are written by hand, which is an enormously expensive and error-prone task. Today there is no lack of tools and approaches to synthesize programs automatically. The problem is that their explanations are mired in a swamp of implementation details and tool-or-approach-specific concepts that makes them difficult to comprehend and compare. In effect, we focus too much on implementation minutia to differentiate results and spend too little time exposing the common abstractions that should unite them.

Many researchers, including myself, are searching for general approaches of automated program development (e.g., [10][30][35]). In this paper, I present ideas that I believe are critical to such an approach, and slowly I am seeing how the pieces fit together. Although a polished integration is far from complete, I want to share with you some of my progress from a personal perspective.

Science has always fascinated me: scientists observe different phenomena and create theories to explain and predict such phenomena. By doing so, the underlying simplicity of Nature is exposed. Newton's laws and Maxwell's unification of magnetism and electricity are classical examples, where mathematics was the language of science. But somewhere in my academic career, I became interested in software design, where a mathematical orientation to design is the exception, rather than the rule.

As I see it, software engineers define structures called programs and use tools to transform, manipulate, and analyze them. Today we see many examples. Object orientation uses methods, classes, and packages to structure programs. Compilers transform source structures into bytecode structures. Refactoring tools map source structures to source structures. And meta-models of *Model Driven Design (MDD)* define the allowable structures of models, and MDD transformations map models to other models for analysis or synthesis. As a community, we are slowly moving toward the paradigm that program design and synthesis is a *computation*. We need a language that brings this fundamental idea to the forefront.

Although not a mathematician, I have come to realize that models of automated software development are intimately related to the language of elementary algebra which provides the essential means to express program design and synthesis precisely. In short, if you look at programming in the right way, it becomes evident that we are using familiar mathematical concepts. Elementary algebra can connect many significant and largely disparate areas of research and, I feel, provides an "architectural langauge" or "architectural framework" to express big-picture concepts in automated software development.

In this paper, I focus on the use of elementary algebra as a language to express fundamental ideas in program design and synthesis. I explain from an informal, algebraic perspective what software engineers do when they create and maintain programs and cover a series of topics (i.e., pieces of the puzzle) that are relevant to automated development, where product lines (i.e., a family of similar programs) are a central focus:

- metaprogramming and product lines,
- testing product lines,
- refactoring product lines, and
- operations for program synthesis.

2 Background

2.1 Program Synthesis and Product Lines

Program synthesis is the idea of programs writing other programs. I view synthesis from a particular perspective: the source text of programs are values (0-ary functions) and transformations are unary (1-ary) functions that map the source of an input program to the source of an output program. The design of a program is an expression (i.e., a composition of functions). Frankly, this is an old idea — it originates from relational query processing of the early 1970s, where the designs of query evaluation programs were written as compositions of relational algebra operations [32].

Recall that relational query processing is one of the great advances that brought databases out of the stone ages to the technologies with which we are familiar today. Instead of manually coding a program to retrieve data, a declarative SQL query is written instead, specifying *what* to retrieve, but not *how.* A parser maps an SQL query to an inefficient relational algebra expression, an optimizer optimizes this expression using algebraic identities, and a code generator maps the optimized expression to an efficient Java or C# program (Figure 1). *The key to the success of relational query processing is that query evaluation programs are defined by relational algebra expressions which can be analyzed and optimized*. It is an example of the paradigm where program design and synthesis is a computation.

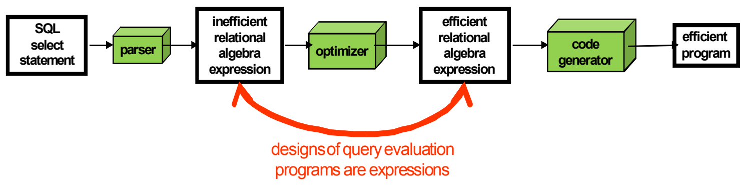

SQL select statement
parser
inefficient relational algebra expression
optimizer
efficient relational algebra expression
code generator
efficient program
designs of query evaluation programs are expressions

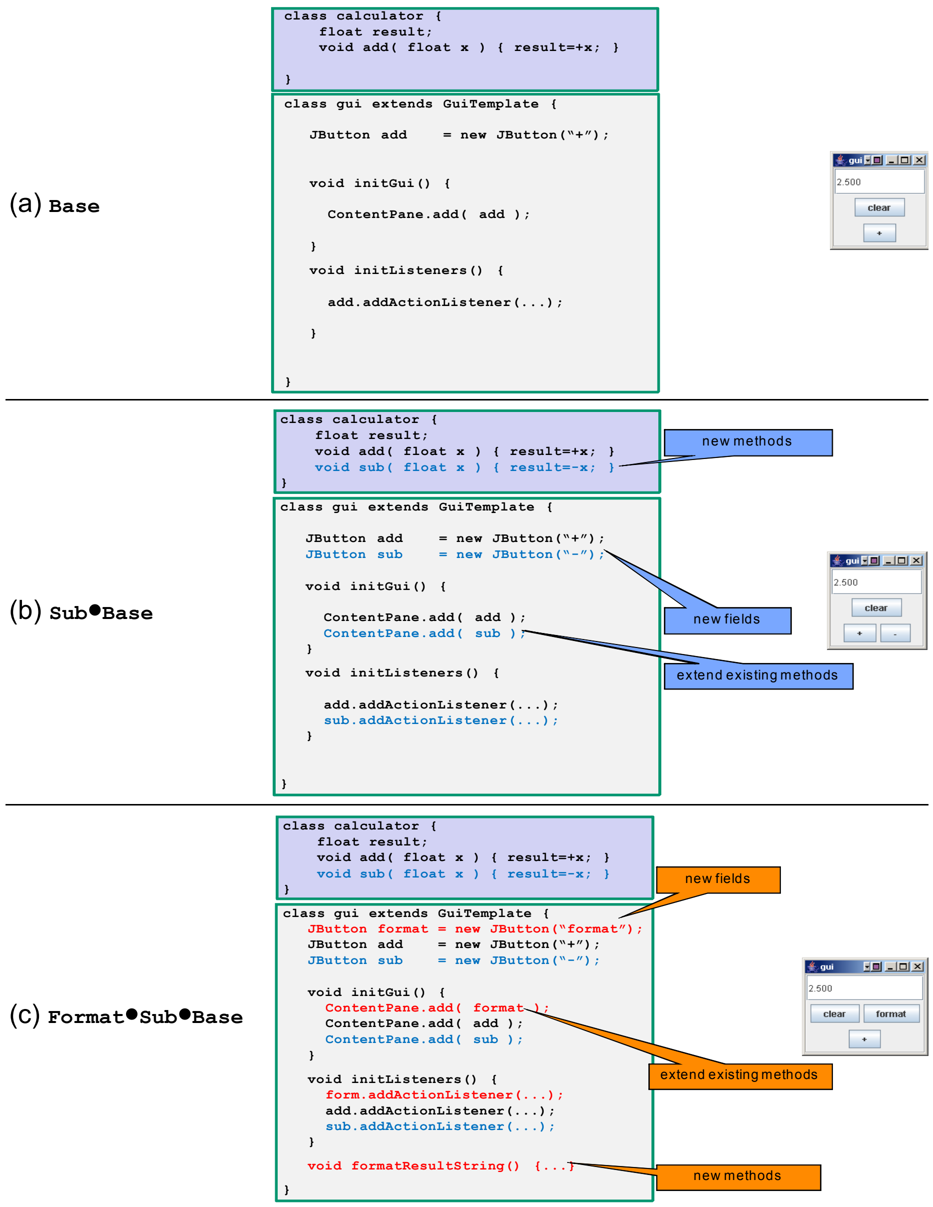
Figure 1. Relational Query Optimization Paradigm

This paradigm generalizes to other domains, where operations are *features* that correspond to increments in program functionality. (A feature roughly corresponds something useful to a customer that some products have while other products don't). A feature is expressed as a function that maps a program without a given functionality to a program with that functionality. Features are a hallmark of *software product lines (SPLs)*, which are families of similar programs. Each program of an SPL is distinguished by its set or composition of features; no two programs have the same composition. One starts with a base program and applies features to progressively elaborate it. Thus the design of a program is an expression (i.e., a composition of features). Expression evaluation is program synthesis, and expression optimization is design optimization [4]. There are many ways to implement features. Popular ways include program transformations [8], aspects [20], mixins [9][13][33], virtual classes [27], refinements [30], and traits [29].

Figure 2 displays an example of a simple calculator and its GUI. Figure 2a shows a **`Base`** calculator which adds numbers. The **`calculator`** class encapsulates the computational functionality and the **`gui`** class implements the GUI. Figure 2b shows the result of composing the **`Sub`** feature, which introduces the subtraction operation to the calculator. Note that the net effect of composing **`Sub`** with **`Base`** is to extend existing methods, add new fields, add new methods, and (although not shown in this example) add new classes. Figure 2c shows the result of composing the **`Format`** feature for controlling the display of computed numbers. As before, adding a feature extends existing methods, adds new fields, adds new methods, and (again, not shown in this example) add new classes. Note that **`Base`** is itself a feature: it adds the rudimentary **`calculator`** and **`gui`** classes to an empty program. By defining a set of features, different compositions of features will yield different programs of a product line. In general, the code modifications that **`Base`**, **`Sub`**, and **`Format`** make are typical of features.

Although the example of Figure 2 is simple, the ideas scale. Twenty years ago, I built extensible database systems exceeding 80K LOC by composing features [2]. Ten years ago, I built extensible Java preprocessors of size 40K LOC by composing features [3]. More recently, I was building the AHEAD Tool Suite, which exceeds 250K LOC, with these same ideas [4]. There are many other people and projects who are doing something similar in creating feature-based product lines for other domains.

In summary, when a product line is created, the building blocks of programs are modules called features that define functions (transformations). A function typically does something very simple: it can add new classes to a program's source, it can extend existing classes with new fields and methods, and it can extend (wrap, advise) existing methods. At least, this is what AHEAD and other tools/languages allow [4][19]. The design of a program in a product line is the task of writing an expression (a composition of functions); the synthesis of the target program's text is expression evaluation.



(a) **Base**

```
class calculator {
     float result;
     void add( float x ) { result=+x; }

}
class gui extends GuiTemplate {

   JButton add     = new JButton("+");


   void initGui() {

     ContentPane.add( add );

   }
   void initListeners() {

     add.addActionListener(...);

   }

}
```

(b) **Sub●Base**

```
class calculator {
     float result;
     void add( float x ) { result=+x; }
     void sub( float x ) { result=-x; }
}
class gui extends GuiTemplate {

   JButton add     = new JButton("+");
   JButton sub     = new JButton("-");

   void initGui() {

     ContentPane.add( add );
     ContentPane.add( sub );
   }
   void initListeners() {

     add.addActionListener(...);
     sub.addActionListener(...);
   }


}
```

(c) **Format●Sub●Base**

```
class calculator {
     float result;
     void add( float x ) { result=+x; }
     void sub( float x ) { result=-x; }
}
class gui extends GuiTemplate {
   JButton format = new JButton("format");
   JButton add    = new JButton("+");
   JButton sub    = new JButton("-");

   void initGui() {
     ContentPane.add( format );
     ContentPane.add( add );
     ContentPane.add( sub );
   }
   void initListeners() {
     form.addActionListener(...);
     add.addActionListener(...);
     sub.addActionListener(...);
   }
   void formatResultString() {...}
}
```

Figure 2. A Calculator and its Graphical User Interface

So the art of program development in product lines is writing functions that implement features, composing these functions, and evaluating the composition.

Although conceptually the idea is simple, it is important to note that conventional programming languages (e.g., Java and C#) have limited facilities to enable programmers to write functions to transform programs. Generics poorly support concepts that are essential to feature-based developoment: (1) *mixins*, a class whose superclass is specified by a parameter, which enables individual classes to be customized, and (2) the scaling of mixins to extend a large number of classes simultaneously [33]. So there is a significant gap between our approach to constructing programs by composing transformations and that provided by conventional programming languages. However, the approach suggests the kinds of extensions that conventional languages will ultimately need. Some of these extensions are described in subsequent sections.

2.2 Simple Algebraic Models of Product lines

Now consider an algebraic description of feature-based product lines. The building blocks of a product line are an empty program and a set of features. The empty program is a value (`0`) and features are unary functions that map an input program (without the given feature) to an output program (that is the input program extended with that feature). The first function applied to `0` provides the infrastructural base code that must be present before any additional feature-related logic can be inserted. The programs of a product line are constructed compositionally by applying features to programs. Different compositions yield different programs of a product line. In effect, *the design of a program is an expression* (i.e., a sequence of unary functions applied to `0`).

On closer inspection, it is well-known that not all compositions of features are meaningful. Product line architects impose constraints on features to limit their compositions only to those that make sense. This is the purpose of a feature diagram, which is a tree-based notation, coupled with constraints, that define the legal combinations of features [12].

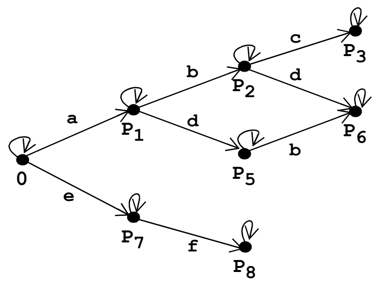
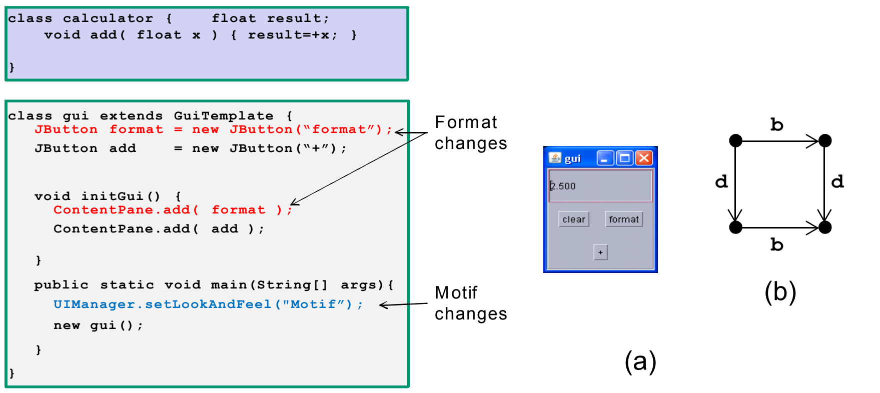


Figure 3. A Category or Product Line

I will not go into the details of feature diagrams, but their net purpose is to express a product line as a directed graph (Figure 3). Objects of the graph are programs in the product line. The initial object is the empty program (`0`). Features are arrows that map an input program to an output program. Each object P_i in the graph defines a domain with one program (the *i*th program of the product line). Arrows are maps (unary functions) that compose. For example, the arrow $P_1 \rightarrow P_3$ can be composed with the arrow $P_3 \rightarrow P_6$ to create an arrow from $P_1 \rightarrow P_6$. Arrow composition is associative. Further, there are identity arrows (maps) for each object, shown as loops in Figure 3. Such a graph is called a *category* [31]. In general, a product line is a category. We will see in later sections how a categorical connection led to recent advances in program testing and synthesis optimization.

The following idea is not part of categories, but it is useful in understanding program synthesis. A traversal from **`0`** to the target program defines a plan (expression) which tells us how to construct that program, step-by-step. In Figure 3, one way to synthesize program $\mathtt{P}_6$ is to apply feature **`a`** to **`0`**, then **`b`**, then **`d`** (i.e., $\mathtt{P}_6\mathtt{=d \bullet b \bullet a \bullet 0}$, where • denotes function composition and **`0`** is a 0-ary function). Such a traversal is commonly called a *makefile* (i.e., do this, then do this, etc., to build $\mathtt{P}_6$).[1]

Figure 3 suggests there can be multiple paths to an object. Another makefile for $\mathtt{P}_6$ is $\mathtt{P}_6\mathtt{=b \bullet d \bullet a \bullet 0}$. In this case, we find an example of *commuting features*, i.e., $\mathtt{d \bullet b = b \bullet d}$, meaning that the order in which features are composed does not matter. Figure 4a depicts such an example for our calculator product line. The features **`Motif`** (giving the calculator GUI a Motif “look-and-feel”) and the **`Format`** feature are commutative: they update disjoint parts of a program, and thus the net effect in which order **`Motif`** and **`Format`** are applied is immaterial.[2]



```
class calculator {     float result;
    void add( float x ) { result=+x; }

}
```

```
class gui extends GuiTemplate {
   JButton format = new JButton(“format”);
   JButton add    = new JButton(“+”);


   void initGui() {
     ContentPane.add( format );
     ContentPane.add( add );

   }

   public static void main(String[] args){
     UIManager.setLookAndFeel("Motif”);
     new gui();
   }
}
```

Figure 4. Commuting Features: **`Motif•Format = Format•Motif`**

Commuting relationships appear in categories as directed rectangles (Figure 4b) called *commuting diagrams*. Visually they represent a simple idea: all paths between two objects in a diagram are equivalent (i.e., each path is a makefile, and different paths yield semantically equivalent makefiles). We will soon see why commuting diagrams are useful.

1. Makefiles also provide an optimization of avoiding the recomputation of stored intermediate results if computation inputs have not changed. This optimization could be applied here, too, but is separate from the point that we are making.
2. I use the notion of *syntactic commutativity*, where the order in which features are composed does not alter the program text. Although semantic commutativity is preferred, one can go quite far with syntactic commutativity in evaluating feature commutativity.

2.3 Program Synthesis

AHEAD [4] is both a methodology and an accompanying set of tools that allow designers to write features as functions that transform the source of an input program to the source of an output program. AHEAD functions have limited capabilities: new entities (e.g. classes) can be added to a program's source, new elements (e.g., fields and methods) can be added to existing entities, and existing elements can be extended (e.g., methods can be wrapped).

AHEAD generalizes the ideas of Section 2.1 by recognizing that programs have multiple representations called *artifacts*. For example, let program P_0 be defined by a state machine specification S_0, its Java source code J_0, and its corresponding bytecode B_0. Program P_0 is represented by a 3-tuple of artifacts $[S_0,J_0,B_0]$. Features are functions that map tuples of input programs to tuples of output programs; each program representation is extended to capture the change that the new feature makes to that representation. For example, suppose feature a maps P_0 to P_1 by extending the original state machine specification S_0 to S_1, the Java code J_0 is extended to J_1, and the bytecode B_0 is extended to B_1. Similarly, feature b maps P_1 to P_2; and feature c maps P_2 to P_3. This mapping is depicted by the horizontal arrows in Figure 5. Features capture the lock-step extension of multiple artifacts, which is the key idea behind feature-based program synthesis.

$$\begin{tikzcd}
P_0 \arrow[r, "a"] & P_1 \arrow[r, "b"] & P_2 \arrow[r, "c"] & P_3 \\
S_0 \arrow[r] \arrow[d] & S_1 \arrow[r] \arrow[d] & S_2 \arrow[r] \arrow[d] & S_3 \arrow[d] \\
J_0 \arrow[r] \arrow[d] & J_1 \arrow[r] \arrow[d] & J_2 \arrow[r] \arrow[d] & J_3 \arrow[d] \\
B_0 \arrow[r] & B_1 \arrow[r] & B_2 \arrow[r] & B_3
\end{tikzcd}$$

Figure 5. Lock-Step Extension of Program Artifacts by Features

Here's the connection of Figure 5 to Figure 3: Let P_0 denote the empty program (with empty source, empty bytecode, and empty documentation). The path $0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3$ in Figure 3 is the path at the top of Figure 5. This linear path is expanded to show the horizontal paths between the three representations of each program, at the bottom of Figure 5. So each path from 0 to a program P_i in Figure 3 corresponds to a mesh of horizontal arrows and program representations in Figure 5.

Model Driven Design (MDD) contributes another fundamental ingredient to automated software development [34][35]. MDD is an increasingly prominent paradigm for program specification and synthesis, and is also based on the idea that a program has multiple representations, but a different terminology is used. A program representation is called a *model*. Functions (a.k.a. transformations) map input models to output models. MDD models are usually just data (e.g., UML class diagrams with no methods), but more generally a model can be any artifact (e.g., a Java class). State machines, source code, and bytecode are examples of models. MDD historically has focussed on the vertical transformations in Figure 5, i.e., the mapping of models of one type to models of another. Integrating MDD (vertical arrows) with product lines (horizontal arrows) is still a topic in its infancy. Returning to vertical arrows, we have a tool **`jak2java:S→J`** that maps state machine specs to Java source, and of course, there is the Java compiler (**`javac:J→B`**) that maps Java source to bytecodes. Although considered tools, **`jak2java`** and **`javac`** are transformations that map one artifact to another.[3]

Feature Oriented Model Driven Design (FOMDD) is a unification of AHEAD and MDD [38][39]. The key idea is to transform arrows that extend high-level artifacts to arrows that extend lower-level artifacts. In Figure 5, a user defines the arrows that map state machines $S_0 \rightarrow S_1$, $S_1 \rightarrow S_2$, and $S_2 \rightarrow S_3$. FOMDD maps these arrows to the corresponding arrows between Java source representations and bytecode representations. That is, FOMDD maps arrow $S_0 \rightarrow S_1$ to arrow $J_0 \rightarrow J_1$ and then arrow $J_0 \rightarrow J_1$ to arrow $B_0 \rightarrow B_1$. The same holds for the other arrows $S_1 \rightarrow S_2$, and $S_2 \rightarrow S_3$. By making all artifact-extension arrows explicit, a commuting diagram of program representations emerges: composing features sweeps out (in this case) the diagram of Figure 5.

A software engineering interpretation of the diagram of Figure 5 is straightforward: start with the object in the upper-left-hand corner (namely the state machine S_0 of program P_0), and derive the object in the lower-right-hand corner (namely the bytecode B_3 of program P_3 — see Figure 6). We know that each path between these two objects is equivalent, in that both derive B_3 from S_0, but they do so in different ways. An immediate observation is that traversing arrows has a cost. When a metric for arrow traversal is defined, the regular geometry of Figure 5a warps to an irregular geometry like Figure 5b. Although all paths produce semantically equivalent results, not all paths are equidistant (meaning that some makefiles are cheaper to execute than others). The shortest path, called a *geodesic*, is the most efficient makefile that synthesizes the target object from the initial object. If all arrows have equal cost as in Figure 6a, any path is a geodesic. However, only the indicated path in Figure 6b is a geodesic.

Note that a "cost metric" need not be a monetary value or execution time; cost may be a measure in production time, peak or total memory requirements, some informal metric of "ease of explanation", or a combination of the above (e.g., multi-objective optimization [43]). The idea of a geodesic is quite general, and should be appreciated from this more general context.

3. More generally, MDD transformations can take *n* input models and produce *m* output models. This is not a problem for us: a function maps a single composite model (which is a tuple of *n* input models) and produces a single composite model (which is a tuple of *n* output models). Projection functions permit access to individual components of a tuple.

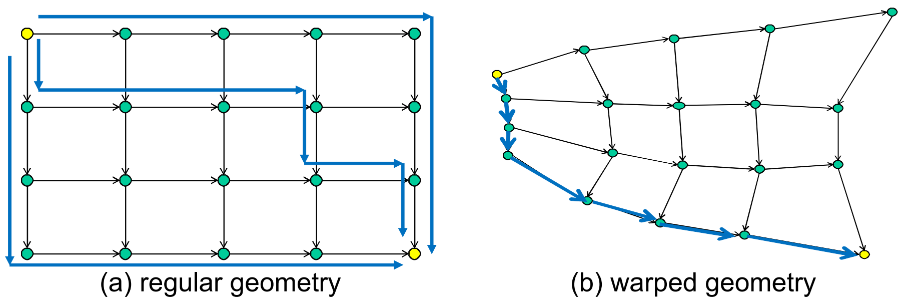


Figure 6. Commuting Diagrams With and Without Cost Metrics

An interesting question is: can geometry warping be used to our advantage? That is, are there interesting problems where geodesics is important? The answer is "yes", and we discuss one such example in the next section.

3 Testing Software Product Lines[4]

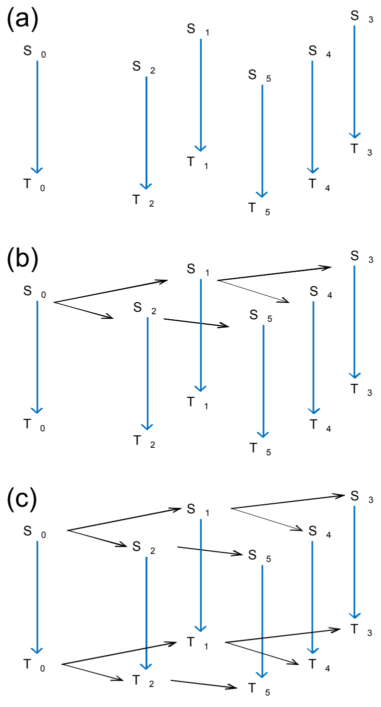


Figure 7. Completion of Commuting Diagrams

Testing software product lines is an important and poorly understood problem. Not only should we be able to generate customized programs given a set of selected features, we also should automatically produce evidence that our generated programs are correct [7][22][23]. In particular, how can we produce tests for every program in a product line? Ideally, our method should be automatic; the manual creation of comprehensive tests scales poorly.

Specification-based testing can be an effective approach for testing the correctness of programs [11][15][18]. The idea is to map a program's specification automatically to test inputs. These inputs are submitted to the program, and the program's response to these tests can be validated automatically using correctness criteria. Figure 7a shows the vertical (derivation) arrows that map specifications of programs $\{S_0 \ldots S_5\}$ of a product line to their tests $\{T_0 \ldots T_5\}$. But we also know that features connect (relate) different program specifications. These are the horizontal arrows in Figure 7b. But elementary mathematics predicts there also must be arrows that connect (relate) generated tests, thus completing the commuting dia-

4. This is joint work with S. Khurshid, E. Uzuncaova, and D. Garcia [40][42].

gram of Figure 7b to yield Figure 7c [31].[5] By completing the diagram, we immediately recognize, for example, that there are multiple ways of producing test $\mathbf{T}_5$ starting from specification $\mathbf{S}_0$. We observed that conventional research follows a particular path: start with the original specification $\mathbf{S}_0$, progressively refine it to $\mathbf{S}_5$, and then derive the test $\mathbf{T}_5$ using some tool. That is, conventional tools and approaches follow particular paths in the diagram of Figure 7c to produce results, but some paths — particularly the paths that refine tests — have not been explored *as we were unaware of them*. The challenge is that it is not at all obvious how to take any path other than the conventional path — we've never taken any other path! Herein lies the potential for geodesics and the "predictions" or generalizations our approach can bring.

Our case study was test generation using Alloy [17][18]. An Alloy specification $\mathbf{S}$ for program $\mathbf{P}$ is written. This specification defines properties (constraints) that the data structures must satisfy. The Alloy analyzer [17][37] maps spec $\mathbf{S}$ to $\mathbf{T}$. To express the mappings of features, we exploit the fact that a feature is an increment in functionality. In principle, we start with a base program $\mathbf{B}$ with Alloy spec $\mathbf{S}_B$. Feature $\mathbf{G}$ has specification $\mathbf{S}_G$ that extends the spec of the base program. When $\mathbf{G}$ is composed with $\mathbf{B}$ to produce program $\mathbf{P}=\mathbf{G}\bullet\mathbf{B}$, let us assume that the composite specification is $\mathbf{S}_P=\mathbf{S}_B\wedge\mathbf{S}_G$ (i.e., the conjunction of the $\mathbf{G}$ and $\mathbf{B}$ specs).[6] The Alloy analyzer translates $\mathbf{S}_P$ into a propositional formula. This formula is solved by a SAT solver yielding $\mathbf{I}_P$. Each solution in $\mathbf{I}_P$ is converted into a test program [24]. The set of all test programs that is produced from $\mathbf{I}_P$ is $\mathbf{T}_P$. The Alloy tool-set has been used to check designs of various applications such as Intentional Naming System for resource discovery in dynamic networks [28], a static program analysis method for checking structural properties of code [36], and formal analysis of cryptographic primitives [26].

Some pragmatic observations: as a specification becomes more complex, finding its solutions tends to become more costly (Figure 8). For example, generating an instance of a linked list with 18 nodes using the Alloy Analyzer takes 41 seconds on average. However, when the specification is refined to that of ordered linked list, computing actual lists of comparable size is exceedingly expensive. In our experiments, we terminated the SAT solver after an hour of computation, unable to find a solution. Clearly, a problem with Alloy is scaling the size of problems it can handle.

product	# of nodes	ave time to generate	ratio
base	18	41s	
ordered•base	18	stopped after 1 hr	> 87x

Figure 8. Scalability of Test Generation

5. The completion of categories, as described above, corresponds to a *pushout* [31].
6. The composition of specifications may not always be this simple, although specification conjunction is both a common assumption [30] and occurrence in actual systems [7].

An elegant way to scale test generation was proposed by Uzuncaova [41]. Instead of solving the entire formula $\mathtt{S}_{\mathrm{P}}$ (as is done conventionally), an alternative is to find a solution $\mathtt{I}_{\mathrm{B}}$ to the base program $\mathtt{S}_{\mathrm{B}}$, and then use $\mathtt{I}_{\mathrm{B}}$ *as a constraint* for solving $\mathtt{S}_{\mathrm{P}}$. That is, start with the solution (tests) of a simpler program, and extend it to a solution (tests) of a more complex program. This procedure is called the *incremental approach*, and it has appealing properties. First, it is sound: any solution of $\mathtt{S}_{\mathrm{P}}$ that can be computed from $\mathtt{I}_{\mathrm{B}}$ is, obviously, a solution of $\mathtt{S}_{\mathrm{P}}$. Second and more interesting, it is complete: any solution to $\mathtt{S}_{\mathrm{P}}$ must embed a solution to subproblem $\mathtt{S}_{\mathrm{B}}$. Thus, by iterating over solutions to $\mathtt{S}_{\mathrm{B}}$, it is possible to enumerate *all* solutions of $\mathtt{S}_{\mathrm{P}}$ (note: some solutions to $\mathtt{S}_{\mathrm{B}}$ may not extend to solutions of $\mathtt{S}_{\mathrm{P}}$, and some $\mathtt{S}_{\mathrm{B}}$ solutions may extend to multiple $\mathtt{S}_{\mathrm{P}}$ solutions). The incremental approach allows us to traverse new synthesis paths that we were previously unaware. The question is: what is the benefit?

Initial experimental results comparing the incremental approach with the conventional approach are encouraging. Figure 9 shows the time for creating tests for a product line of lists (a standard example of researchers using the Alloy analyzer). For some experiments, the conventional approach was faster. The reason is that the composite predicates were simple enough to solve directly — it was overkill to partition them into elementary predicates, solve the simpler 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 20× faster. The reason is that it is easier to find solutions to simple predicates and to extend those solutions.

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 [21] for details), in principle, the idea is clear for the way Alloy specifications are composed. Figure 10 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 ($\mathtt{T}_2$) is visualized by a 3-dimensional commuting

subject	product	speed-up
Doubly-Linked List (scope=8)	*base*	n/a
	size • *base*	1.33×
	double • *base*	0.80×
	ordered • *base*	17.39×
	size • *double* • *base*	0.54×
	ordered • *size* • *base*	30.35×
	ordered • *double* • *base*	10.67×
	ordered • *size* • *double* • *base*	24.63×
Intentional Naming System (scope=16)	*base*	n/a
	attr-val • *base*	0.35×
	label • *attr-val* • *base*	14.53×
	record • *label* • *attr-val* • *base*	9.56×

Figure 9. Conventional v.s Incremental Test Generation

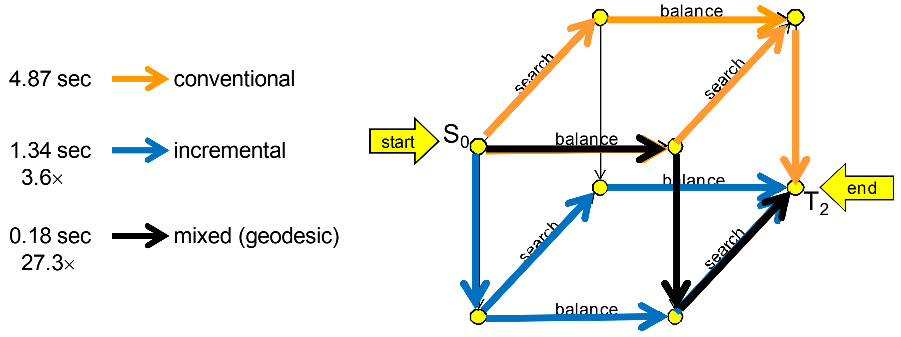


Figure 10. Geodesic in a Commuting Diagram

diagram. Note that the conventional and incremental approaches correspond to particular paths in this diagram. We evaluated all possible paths through this cube.

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. Further work by Uzuncaova introduced a constraint prioritization approach that can assist in identifying an optimal path for test generation; details of this approach are described elsewhere [41].

Although this line of work (e.g., following novel paths to synthesize program artifacts) is in its infancy, initial results are encouraging. Elementary mathematics tells us ways of generating results efficiency that we didn't have before — what we are doing above is exploiting geometry warping. For more details, see [42]. For examples of using geodesics for optimizing the synthesis of programs, see [38][39].

4 Refactoring Product Lines

A *refactoring* is a disciplined technique (a.k.a. transformation) for restructuring a body of code that changes its structure but not its behavior [14]. There are many common refactorings in use in the *object-oriented (OO)* programming: move field (from one class to another), delete method (usually done when no references to the method exist), change argument type (i.e., replacing an argument type with its supertype), replace method call (with another that is semantically equivalent in the same class), and so on. An interesting question is: how do refactorings affect a product line? What happens a feature is refactored, say by moving a field or method from one class to another? Not surprisingly, little is known about this subject. In this section, I present conjectures on possible directions of research and how our approach/language illuminates this topic.

A common design technique in product lines is to superimpose the OO class diagrams of all programs. Doing so defines a class diagram of a "master plan" for all programs in the SPL. It encourages a standard meaning and naming convention for all classes and their members that appear in any program of a product line. Stated differently, a "master plan" avoids the complexity and confusion that would arise if inconsistent meanings and names are used for the same method (e.g., **m()** means θ in program P_1, but is named **n()** in program P_2, and means ¬θ in program P_3). Such inconsistencies would make a product line incomprehensible to engineers who are responsible for maintaining and extending it. Standardization of meanings and names is a common way to control complexity in SPLs and in many other engineering disciplines [1].

Figure 11a depicts a master plan. The **black** feature has classes **A** (members **x,y**), and **B** (members **r,s**). The **orange** feature adds class **C** (members **u,v**) and member **t** to **B** and **z** to **A**. The **blue** feature adds class **D** (members **m,n**). And the **red** feature adds class **E** (members **i,j,k**) and member **w** to **C**. Eliminating unwanted features yields the class diagram of a program in the master plan's product line.

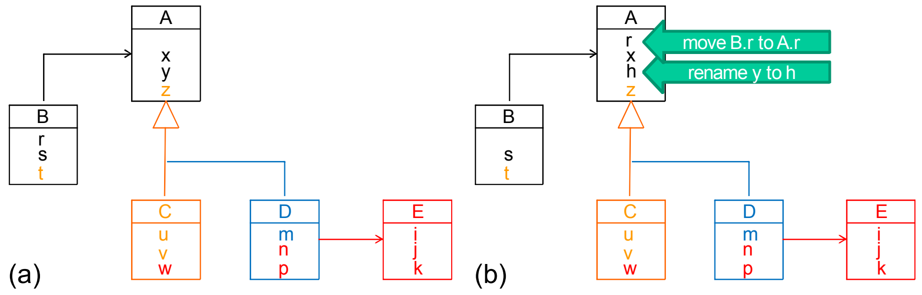


Figure 11. Refactoring Master Plans of SPLs

Refactoring a feature can involve any standard OO refactoring: members can be renamed, members can be moved from one class to another, etc. There are also non-standard refactorings that are feature-specific, such as moving members from one feature to another. In general, refactoring a feature alters many programs of a product line. As an example, if member **y** in class **A** is renamed to **h**, then all programs of the product line that use the **black** feature will see this renaming (Figure 11b). The same holds for moving method **r** in class **B** to class **A** (Figure 11b): all programs of the product line that use the **black** feature will see these changes.

Here is a working hypothesis (conjecture): refactoring an SPL is the same as refactoring one huge program where typically not all pieces of this program are present in any one member of this SPL. Composition of features is modeled by a projection of this "huge" program that eliminates unneeded features. So by refactoring a single "huge" program, an entire product line is refactored.

To better understand the refactoring of features, consider Figure 12a. Suppose the **`black`** feature maps the empty program (**`0`**) to program **`P`**$_1$. Any change to **`black`** that we considered (e.g., renaming **`y`** or moving **`B.r`** to **`A.r`**) will be visible to any program "downstream" (meaning any program that is derivable from) **`P`**$_1$. Any program that does not use the **`black`** feature, such as **`0`**, **`P`**$_2$, and **`P`**$_7$, will be oblivious to this change.

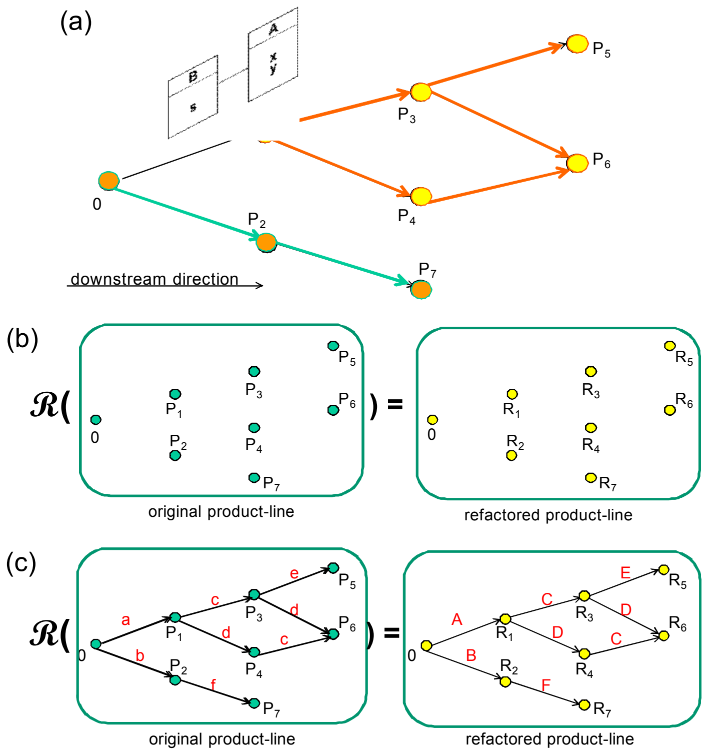


Figure 12. Refactoring Product Lines

So when a refactoring $\mathcal{R}$ is applied to a feature, it potentially transforms every program in a product line. That is, $\mathcal{R}$ maps each program of the original product line to a corresponding and unique program in the refactored project line (Figure 12b). In effect, $\mathcal{R}$ defines the object-to-object mappings from the original category (product line) to the new category (the refactored product line). But looking closer, we recognize that programs of a product line are *not* stored — they are *computed* by composing features. What is actually being refactored are the *arrows* (the modules that implement individual features). So a product line refactoring actually maps both objects and arrows of a category (product line) such that the connectivity properties of the original category (product line) are preserved. Stated differently, a refactoring is a structure preserving map between two categories. This concept is known as a *functor* in category theory [31].[7] The functors that frequently arise in feature-based development are maps between isomorphic categories (i.e., categories that have the same shape, but possibly different labels for corresponding objects and arrows). We call such functors *manifest*.

We have seen several examples of manifest functors already in this paper. Each tuple of Figure 5 defines a category of program artifacts (the **`jak2java`** tool maps a state machine spec to its Java code counterpart, **`javac`** maps Java source to bytecodes). Features define manifest functors from one tuple to another. Figure 7a defines a manifest functor from a product line of program specifications to a product line of program tests. Alloy tools implement the object-to-object mappings of this functor.

I conjecture that features, MDD transformations, and refactorings can be unified in the following way. Consider Figure 13. Starting with a state machine specification of program $\mathbf{P}_0$, we want to derive the bytecodes ($\mathbf{B}_1$) for a refactored program $\mathbf{P}_1$. A conventional way is to refine $\mathbf{S}_0$ by applying a feature, and then refactor the state machine (e.g., renaming states), and then derive its bytecode implementation ($\mathbf{B}_1$). This corresponds to the "upper-perimeter" path of the cube in Figure 13. Alternatively we might immediately derive the bytecode implementation of $\mathbf{S}_0$, refactor the bytecode, and then apply the corresponding (refactored) bytecode feature to produce $\mathbf{B}_1$. This corresponds to the "bottom-perimeter" path of the cube in Figure 13. A pragmatic reason for this alternative path is that one does not have to expose the state machine specifications (or their source refinements) to users. If a product line comes with a set of binary (not source) features that can be composed and refactored, the *intellectual property (IP)* of the original state machines may be better preserved. This certainly is the case for conventional COM components and proprietary Java libraries which are typically distributed in binary form for, among many reasons, increased IP protection.

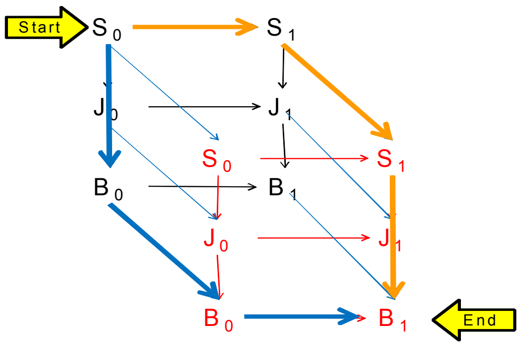


Figure 13. Unifying Refactorings, MDD, and Features

Commuting diagrams, such as Figure 13, suggest how elementary mathematics can neatly tie together basic concepts in feature-based product lines, transformations in MDD, and refactorings. But much more work is needed to (a) demonstrate this and (b) recognize the technical and educational benefits in doing so. This is a subject of ongoing work.

7. A *functor* from category `C1` to category `C2` is an embedding of `C1` into `C2` such that `C1`'s connectivity properties are preserved [31].

5 Operations for Program Synthesis[8]

As mentioned earlier, AHEAD defines features as functions that map tuples to tuples. In my informal conversations with mathematicians many years ago, a question arose frequently: can feature compositions be modeled by a vector space? Of course, I had no answer and only recently began thinking about it and its implications.

Informally, a *vector space* is a collection of tuples called *vectors*, where vectors can be added and scaled. Formally, a vector space satisfies a number of basic axioms, such as vector addition:

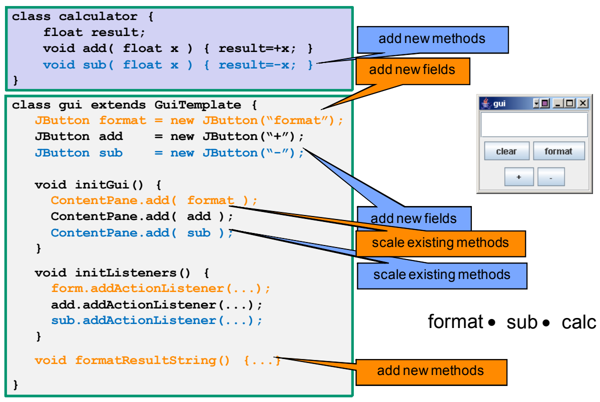
- is commutative: $\forall x,y \in V$: $x+y=y+x$
- is associative: $\forall x,y,z \in V$: $(x+y)+z=x+(y+z)$
- and has an additive identity: $\forall x \in V$: $0+x=x$

where V is the set of all vectors and $0 \in V$ is the zero vector. Further, vectors can be scaled by multiplication:

- scalar multiplication: $\forall m \in M$: $m \cdot [a,b,c]=[m \cdot a,\ m \cdot b,\ m \cdot c]$
- scalar multiplication distributes over vector addition: $\forall m \in M$ and $\forall x,y \in V$: $m \cdot (x+y)=m \cdot x+m \cdot y$

where M is the set of all scalar multipliers and $m \cdot x$ means scale vector x by m. Of course, my immediate reaction (like yours no doubt) is: what does this have to do with software development? But on further thought, I realized that my questions should have been: Is there an addition operation in software development? And is there a scaling operation? To my surprise, the answer to both questions is "yes". Every feature that I have built with AHEAD has both operations, but I failed to recognize them.

Recall an earlier example, which I reproduce in Figure 14. Note that when a feature is composed, we see an addition operation in action: a feature can add new classes and add new members to existing classes. The order in which classes/members are added is immaterial (i.e., addition is commutative and associative), and adding



```
class calculator {
    float result;
    void add( float x ) { result=+x; }
    void sub( float x ) { result=-x; }
}
class gui extends GuiTemplate {
   JButton format = new JButton("format");
   JButton add    = new JButton("+");
   JButton sub    = new JButton("-");

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

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

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

Figure 14. Addition and Modification of Java Source

8. This is joint work with D. Smith [5].

nothing to a program yields that program (i.e., addition has an additive identity, namely `0`, the empty program).

There is also a modification operation, which is a form of scaling: a feature can extend existing methods with new code. There are many ways in which code modifications can be expressed, but all satisfy the properties of scalar multiplication. For example, a transformation is a `(pattern, rewrite)` pair [8]: when the `pattern` is found in source code, it is modified according to the `rewrite`. A transformation is applied to all components of a program (i.e., it matches the idea above of scalar multiplication where the modification is applied to all components of a vector). One can recognize these ideas in *Aspect Oriented Programming (AOP)*: AOP advice is a `(pointcut, modifier)` pair: the `pointcut` identifies patterns in program execution, and the `modifier` is extra code that is to be run when that pattern occurs during run-time. Although AOP advice is understood in terms of extending program executions, it is well-known that AOP compilers weave advice statically, which can be conceptualized by transformation `(pattern, rewrite)` pairs [25].

To make this concrete, consider the following example. Figure 15a shows a `Base` buffer whose value can be set. Figure 15b shows the `Restore` feature (as expressed in AspectJ, which we assume a minimal familiarity [20]), that allows one to restore the previous contents of a buffer. Figure 15c shows the composition of `Restore•Base`.

```
class buffer {
  int buf=0;

  void set(int i)
  { buf=i; }
}
```
(a)

```
aspect Restore {
  int buffer.back=0;

  void buffer.restor()
  { buf=back; }

  before():
    execution(set(int))
    { back=buf; }
}
```
(b)

```
class buffer{
  int buf=0;
  int back=0;

  void set(int i)
  { back=buf; buf=i; }

  void restor()
  { buf=back; }
}
```
(c)

Figure 15. `Base`, `Restore`, and `Restore•Base`

Let's see how we can express this design algebraically. We model a class by a tuple, one component per possible member. The tuple for class `buffer` (Figure 15a) is `Base()=[buf,set,0,0]`, where `buf` denotes the Java declaration of the `buf` variable and `set` denotes the Java declaration of the `set()` method. The extra zeros (`0`) mean that the `restor` and `back` members are presently undefined.

We model the `Restore` feature as a unary function that takes a tuple `v` as input and produces a tuple as output:

```
    Restore(v) = [0,0,back,restor] + Δset·v
```

Let's see what the above means. The `Restore` feature adds members `back` and `restor` to class `buffer`. This is expressed by the tuple `[0,0,back,restor]`. The before advice is represented by `Δset`, which is to be applied to the input class `v`. To compose `Base` with `Restore`, we evaluate `Restore•Base`:

```
Restore•Base
= [0,0,back,restor] + Δset·[buf,set,0,0]        // substitution
= [0,0,back,restor] + [Δset·buf,Δset·set,Δset·0,Δset·0]
                                                // scalar mult.
```

The above expression can be simplified by noting that `Δset` only affects the `set()` member (component); it has no effect on the other members (as `Δset` does not capture any of their join points). Let `set′` denote the Java definition of the `set()` method in Figure 15c. Simplifying:

```
= [0,0,back,restor] + [buf,Δset·set,0,0]        // simplify
= [0,0,back,restor] + [buf,set′,0,0]            // substitution
= [buf,set′,back,restor]                        // addition
```

Note that the resulting tuple `[buf,set′,back,restor]` expresses the class `buffer` in Figure 15c. The last step in a computation is to transform this tuple into its source code representation (Figure 15c).

Here's how to understand this calculation in a more general setting: given an input feature expression (e.g., `Restore•Base`), a compiler will inhale the code of each feature; convert the code into an arithmetic expression; evaluate, simplify, and possibly optimize the feature expression; and translate the resulting tuple into the output program, just as we did above. In effect, I foresee that feature-based compilers will become program calculators that use simple algebraic rewrites to optimize program synthesis. In effect, this is what AHEAD is doing now, except at a much finer level of granularity.

In [5][6], I show how these ideas can be taken further. Refactorings are operators that map expressions to expressions. So the idea that engineers manipulate programs algebraically by tools and compilers is given a more algebraic foundation. Of course, much more work is necessary, but hopefully you get the idea.

Having said this, there are clear mismatches in program development and vector spaces. Scaling (modification) of source code is highly non-uniform. Only selected methods are modified by an advice in AOP. Source code does not seem to have an additive inverse. Classes and class members can be deleted, but there does not appear to be the notion of a "negative" or "anti" method, which (when added to its positive counterpart) annihilates that method. Further, it is debatable whether modifiers (advice) belong to the same type of elements as method definitions and fields.

Although the analogy with vector spaces is at best suggestive, it is still useful. Making explicit the operations of addition, subtraction, and modification offers a simple language to explain complex processes involving program refactoring and relating different feature-based programming concepts by focussing on their similarities, rather than their implementation differences [6][25].

6 Conclusions

Software engineers define structures called programs that evolve though additions (of classes, methods, fields), deletions (removal of classes, methods, fields), and transformations (adding features and refactoring). The language of simple mathematics can be used to describe these processes in an understandable and uniform way, and has both pedgogical and practical benefits, such as revealing new ways to synthesize artifacts.

Ultimately, however, it requires us to think differently. Software development may be an ad hoc practice in general, but the automated development of software in well-understood domains should not be. It requires us *not* to think in terms of monolithic designs, but rather in terms of changes to designs, and composing (or rather integrating) a sequence of changes to produce complete designs. That is really what we do when we build and modify programs incrementally, although we don't normally think of program construction in this way.

Clearly there is a lot more to do. Using mathematics to express the essense of automated software development is, in my opinion, a first step toward principled automated software engineering. It will tell us on how to think about program construction in a structured and non-ad-hoc way. It is clear that many ideas are being reinvented over and over again: this is not accidental; it is a symptom or characteristic that what we are doing is part of a larger paradigm that we are only now beginning to understand. Doing so will lead to better design and program construction techniques, better tools and languages, and better design methodologies. And it may also lead the way to deeper applications of mathematics to the construction and synthesis of programs with assured or verified properties.

Acknowledgements. I appreciated the help and patience of Prof. Egon Börger by allowing me to take my time in writing this paper. I also appreciated the helpful and insightful comments of the referees.

7 References

[1] ASME web site. `http://www.asmesolutions.org/Energy/Nuclear.cfm`

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

[3] D. Batory, B. Lofaso, and Y. Smaragdakis. "JTS: Tools for Implementing Domain-Specific Languages". *ICSR 1998*.

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

[5] D. Batory and D. Smith. "Finite Map Spaces and Quarks: Algebras of Program Structure". University of Texas at Austin, Dept. of Computer Sciences, TR-07-66.

[6] D. Batory. "Program Refactorings, Program Synthesis, and Model-Driven Design". *ETAPS-CC 2007*.

[7] D. Batory and E. Börger. "Modularizing Theorems for Software Product Lines: The Jbook Case Study". To appear, *JUCS*.

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

[9] G. Bracha and W. Cook. “Mixin-Based Inheritance”. *OOPSLA and ECOOP 1990*.

[10] M. Bravenboer, K.T. Kalleberg, R. Vermaas, and E. Visser. “Stratego/XT 0.17. A Language and Toolset for Program Transformation”. *Sci. of Computer Programming*, 2008

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

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

[13] M. Flatt, S. Krishnamurthi, and M. Felleisen. “Classes and Mixins”, *POPL 1998*.

[14] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2005.

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

[16] D. Jackson, I. Schechter, and I. Shlyakhter. “ALCOA: The Alloy Constraint Analyzer”. *ICSE 2000*.

[17] D. Jackson. “Alloy: A Lightweight Object Modeling Notation”. *ACM TOSEM*, April 2002.

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

[19] C. Kästner, S. Apel, and M. Kuhlemann. “Granularity in Software Product Lines”. *ICSE* 2008.

[20] G. Kiczales, et al. “An Overview of AspectJ”. *ECOOP 2001*.

[21] C.H.P. Kim, C. Kästner, and D. Batory. “On the Modularity of Feature Interactions”, submitted 2008.

[22] S. Krishnamurthi and K. Fisler. “Modular Verification of Collaboration-Based Software Designs”, *FSE 2001*.

[23] S. Krishnamurthi, K. Fisler, and M. Greenberg. “Verifying Aspect Advice Modularly”, *ACM SIGSOFT 2004*.

[24] S. Khurshid, “Generating Structurally Complex Tests from DeclarativeConstraints”, Ph.D. Thesis, MIT EECS, 2003.

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

[26] A. Lin, M. Bond, and J. Clulow. “Modeling Partial Attacks With Alloy”. *Security Protocols Workshop (SPW)*. April 2007.

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

[28] D. Marinov and S. Khurshid. “TestEra: A Novel Framework for Automated Testing of Java Programs”. *ASE 2001*.

[29] E.R. Murphy-Hill, P.J. Quitslund, and A.P. Black, “Removing Duplication from java.io: A Case Study Using Traits”, *OOPSLA 2005*.

[30] 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.

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

[32] 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*.

[33] Y. Smaragdakis and D. Batory. “Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs”. *ACM TOSEM*, April 2002.

[34] T. Stahl and M. Voelter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[35] J. Sztipanovits, “Generative Programming for Embedded Systems”, *GCSE 2002*.

[36] M. Taghdiri. "Inferring Specifications to Detect Errors in Code". *ASE 2004*.

[37] E. Torlak and D. Jackson. "Kodkod: A Relational Model Finder". *TACAS 2007*.

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

[39] S. Trujillo, D. Batory, O. Diaz. "Feature Oriented Model Driven Development: A Case Study for Portlets". *ICSE 2007*.

[40] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. "A Specification-based Approach to Testing Software Product Lines", Poster Paper, *ACM SIGSOFT 2007*.

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

[42] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory, "Testing Software Product Lines Using Incremental Test Generation", submitted 2008.

[43] Wikipedia, *Multiobjective optimization*, `http://en.wikipedia.org/wiki/Multiobjective_optimization`