Presented at Generative Programming and Component Engineering (GPCE) 2004, Vancouver, British Columbia.

Automatic Remodularization and Optimized Synthesis of Product-Families

Jia Liu and Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{jliu, batory}@cs.utexas.edu

Abstract. A *product-family* is a suite of integrated tools that share a common infrastructure. Program synthesis of individual tools can replicate common code, leading to unnecessarily large executables and longer build times. In this paper, we present remodularization techniques that optimize the synthesis of product-families. We show how tools can be automatically remodularized so that shared files are identified and extracted into a common package, and how isomorphic class inheritance hierarchies can be merged into a single hierarchy. Doing so substantially reduces the cost of program synthesis, product-family build times, and executable sizes. We present a case study of a product-family with five tools totalling over 170K LOC, where our optimizations reduce archive size and build times by 40%.

1 Introduction

Compositional programming and automated software engineering are essential to the future of software development. Arguably the most successful example of both is *relational query optimization (RQO)*. A query is specified in a declarative domain-specific language (SQL); a parser maps it to an inefficient relational algebra expression; the expression is optimized; and an efficient query evaluation program is generated from the optimized expression. RQO is a great example of *automatic programming* — transforming a declarative specification into an efficient program, and *compositional programming* — a program is synthesized from a composition of algebraic operators.

Our research is *Feature Oriented Programming (FOP)* which explores feature modularity and program synthesis [3]. We believe that FOP generalizes the paradigm exemplified by RQO so that compositional programming and automated software development can be realized in any domain. FOP supports the paradigm of mapping declarative specifications (where users specify the features they want in their program) to an actual implementation. This is possible because programs are synthesized by composing implementations of the required features. The novelty of FOP is that it models software domains as algebras, where each feature is an operation. Particular programs that can be synthesized are compositions of operations.

The hallmark of the RQO paradigm is the ability to optimize algebraic representations of programs using identities that relate domain operators. The commutability of joins and the distributivity of project over joins are examples in relational algebra. In this paper, we demonstrate an interesting example of automatic algebraic optimization and

reasoning to accomplish the remodularization and optimized synthesis of product-families.

A *product-family* is a suite of integrated tools that share a common code base. A Java *Integrated Development Environment (IDE)* is an example: there are tools for compiling, documenting, debugging, and visualizing Java programs. Engineers perform two tasks when these tools are designed and coded *manually*: (1) they create and implement a design for each tool and (2) integrate each design with the design of other tools to minimize code replication. A paradigm for automated software development achieves the same result in a similar way. Individual tools are synthesized from declarative specifications. This allows, for example, multiple tools to be developed simultaneously because their implementations are completely separate. This is possible because the common code base for these tools is replicated. However, to achieve the "optimized" manual design where common code is not replicated requires an *optimization* that (1) breaks the modular encapsulations of each synthesized tool and (2) identifies the infrastructure shared by all tools and factors it out into common modules. So not only must tools be synthesized automatically, so too must the post-synthesis remodularization optimization be done *automatically*.

In this paper, we present two optimizations that remodularize tool applications automatically. The first resembles the common practice of extracting shared classes into a common library, but is more efficient with algebraic analysis than brute-force file comparisons. The second merges isomorphic class inheritance hierarchies into a single hierarchy, which delivers better results than extracting shared classes. Both optimizations substantially reduce the size of executables, the cost of tool synthesis, and product-family build times.

We present a case study of a product-family with five tools totalling over 170K LOC. Our optimizations reduce its generation and build times by 40%. Although the percentage reductions are specific to the case study, the techniques we present are general and are applicable to product-families in arbitrary domains.

2 FOP and AHEAD

AHEAD (Algebraic Hierarchical Equations for Application Design) is a realization of FOP based on step-wise refinement, domain algebras, and encapsulation [1].

2.1 Refinements and Algebras

A fundamental premise of AHEAD is that programs are constants and refinements are functions that add features to programs. Consider the following constants that represent base programs with different features:

```
f       // program with feature f
g       // program with feature g
```

A *refinement* is a function that takes a program as input and produces a refined or feature-augmented program as output:

```
i(x)    // adds feature i to program x
j(x)    // adds feature j to program x
```

A multi-featured application is an *equation* that corresponds to a refined base program. Different equations define a family of applications, such as:

```
app1 = i(f)     // app1 has features i and f
app2 = j(g)     // app2 has features j and g
app3 = i(j(f))  // app3 has features i, j, f
```

Thus, the features of an application can be determined by inspecting its equation.

An *AHEAD model* or *domain model* is an algebra whose operators are these constants and functions. The set of programs that can be synthesized by composing these operators is the model's *product-line* [2].

2.2 Encapsulation

A base program typically encapsulates multiple classes. The notation:

```
P = { A, B, C }
```

means that program **P** *encapsulates* classes **A**, **B**, and **C**.

When a new feature **R** is added to a program **P**, any or all of the classes of **P** may change. Suppose refinement **R** modifies classes **A** and **B** and adds class **D**. We write that **R** encapsulates these changes:

```
R = { ΔA, ΔB, D }
```

The refinement of **P** by **R**, denoted **R(P)** or **R•P**, composes the corresponding classes of **R** and **P**. Classes that are not refined (**C**, **D**) are copied:

```
R•P  = { ΔA, ΔB, D } • { A, B, C }
     = { ΔA•A, ΔB•B, C, D }                    (1)
```

That is, class **A** of program **R•P** is generated by composing **ΔA** with **A**, class **B** of **R•P** is **ΔB•B**, etc. **(1)** illustrates the *Law of Composition*, which defines how the composition operator • distributes over encapsulation [1]. It tells us how to synthesize classes of a program from the classes that are encapsulated by its features (e.g., base programs and refinements).

2.3 AHEAD Tool Suite (ATS)

ATS is a product-family that implements the AHEAD model. All ATS source files are written in the Jak (short for Jakarta) language; Jak is Java extended with refinement declarations, metaprogramming, and state machines. Jak programs are indistinguishable from Java programs, except those that express refinements and state machines.

As examples of Jak source, Figure 1a shows class **C** (which is identical to its Java representation). Figure 1b shows a refinement **ΔC** which adds variable **y** and method **h** to **C**. In gener-

```
class C {
  int x;
  void g() {..}
}
```

(a) **C**

```
refines class C {
  int y;
  void h() {..}
}
```

(b) **ΔC**

```
class C {
  int x;
  int y;
  void g() {..}
  void h() {..}
}
```

(c) **ΔC•C**

Figure 1. Class Definition, Refinement, and Composition

al, a class refinement can add new data members, methods or constructors to a class, as well as extend existing methods and constructors. The composition of ∆**C** and **C**, denoted ∆**C(C)** or ∆**C•C**, is shown in Figure 1c; composition merges the changes of ∆**C** into **C** yielding an updated class. See [1] for more details about Jak specifications.

In general, an AHEAD constant encapsulates a set of classes (as in Figure 1a). An AHEAD function encapsulates a set of classes and class refinements (as in Figure 1a-b). That is, an AHEAD function refines existing classes and can add new classes that can be subsequently refined. Thus, an AHEAD function typically encapsulates a *cross-cut*, meaning that it encapsulates fragments (refinements) of multiple classes. Composing AHEAD constants and functions yields packages of fully formed classes [2]. As AHEAD deals with cross-cuts, it is related to Aspect-Oriented Programming. We explore this relationship in Section 4.

ATS has five core tools that transform, compose, or introspect Jak files:

- **jak2java** translates a Jak file to its corresponding Java file,
- **jampack** and **mixin** are different implementations of the composition operator for Jak files,
- **unmixin** uncomposes a composed Jak file, and
- **mmatrix** provides code browsing capabilities.

These tools are fairly large, the sum of their individual sizes exceeds 170K LOC in Java.

ATS has been bootstrapped, so that each ATS tool has its own AHEAD equation and the tool itself is synthesized from this equation. The AHEAD synthesis process expands a tool's equation **T = j•i•...•k** using (1) so that each tool is equated with a set of expressions **{**$\mathbf{e}_1$**,** $\mathbf{e}_2$**,** $\mathbf{e}_3$**,** …**}**, one expression $\mathbf{e}_i$ for each class that the tool encapsulates:

T = j•i•...•k
= {$\mathbf{e}_1$**,** $\mathbf{e}_2$**,** $\mathbf{e}_3$**,** …**}**

Evaluating each of the $\mathbf{e}_i$ generates a particular file of the target tool.

2.4 Optimizing Product-Families

Although different tools or members of a product-family are specified by different feature sets, tools can share features, and thus share a significant code base. In object-oriented systems this corresponds to shared classes or shared methods. For example, when a tool is synthesized from an AHEAD equation, all the classes that comprise that tool — both common and tool-specific — are generated. Thus, if there are *n* tools, each common class will be generated *n* times. Not only does this result in longer build times, it also leads to code duplication in tool executables, since every tool has its own package where the classes are replicated.

For example, each ATS tool performs an operation — composition, translation, or introspection — on a Jak file. This means each tool shares the same parser (which comprises 4 classes). ATS tools also share the same parse tree classes, which are represented by an inheritance hierarchy of *abstract syntax tree (AST)* nodes.

Both the parser and the inheritance hierarchy are generated from the grammar of the Jak language. Parser generation from a grammar is well-known; less well-known is our algorithm for synthesizing the AST inheritance hierarchy. Consider Figure 2a which shows grammar productions **`Rule1`** and **`Rule2`**, where **`C1`**, **`C2`** and **`C3`** are terminals. From this, the inheritance hierarchy of Figure 2b is inferred. Each production and pattern is mapped to a distinct AST class. Production-pattern relations are captured by inheritance.

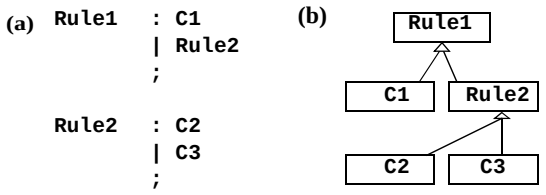

Figure 2. Grammars and Inheritance Hierarchies

The generated AST hierarchy corresponds to a single AHEAD feature that is shared by all tools. Its classes have bare-bones methods and data members that allow ASTs to be printed and edited. These classes are subsequently refined with new data members and methods that implement tool-specific actions.

The Jak grammar has 490 rules, so the inheritance hierarchy that is generated has 490 classes. After refinement, over 300 of these classes are identical across Jak tools; those that are different result in the different actions per tool. Clearly, such redundancies increase the generation time and build time for ATS, and lead to larger executables. In the following sections, we present two ways to automatically eliminate such redundancies.

2.5 Shared Class Extraction (SCE)

A common practice to eliminate redundancies is to factor files into tool-specific and tool-shared (or common) packages. For example, in ATS tools domain-knowledge tells us that the common parser files can be manually extracted into a common package, but these are just 4 files. The vast majority of the common classes reside at various locations in the AST hierarchy, and which files are common depends on the tools being considered. A brute-force way to identify common classes is by "diffing" source files. This way one can only recognize identical files *after* all files have been composed and generated. To make the problem worse, the corresponding composed Java files have different **`package`** declarations, so the **`diff`** utility that is used in file comparison would have to ignore differences in package names. This involves a lot of unnecessary work.

Ideally, we want a procedure that automatically identifies all common files *before* they are generated, and extracts them into a common library. *No* knowledge about product-family design (e.g., common parser) should be needed for this optimization. This ideal can be achieved using algebraic reasoning.

Solution. We introduce an operator ⊕ to compute the common classes from multiple tools:

$$\texttt{Common} = \texttt{T}_1 \oplus \texttt{T}_2 \oplus \ldots \oplus \texttt{T}_n \qquad (2)$$

The operator ⊕ is both associative and commutative, meaning that the order in which tools are applied does not matter:

$$(\texttt{T1} \oplus \texttt{T2}) \oplus \texttt{T3} = \texttt{T1} \oplus (\texttt{T2} \oplus \texttt{T3}) \qquad (3)$$

$$\mathbf{T1} \oplus \mathbf{T2} \quad = \quad \mathbf{T2} \oplus \mathbf{T1} \qquad (4)$$

Each tool of ATS is defined by a set of equations, one equation for each artifact (Jak file, grammar file) that the tool encapsulates. Suppose tools **T1** and **T2** both encapsulate files **x**, **y**, **z**, and **w**. Further suppose the equations for these files for **T1** are:

```
x = x3•x2•x1
y = y2•y1
z = z1
w = w2•w1
```

and the equations for these files for **T2** are:

```
x = x3•x2•x1
y = y3•y1
z = z1
w = w3•w1
```

Without generating the files, we conclude from these definitions that files **y** and **w** are different across the two tools, because their equations differ. On the other hand, file **x** in **T1** is the same as in **T2**, because both equations are identical[1]. By the same reasoning, file **z** is also identical in **T1** and **T2**. Thus, files **x** and **z** are shared by **T1** and **T2** and can be placed in a shared package. Hence:

```
Common= T1 ⊕ T2
      = { x3•x2•x1, y2•y1, z1, w2•w1 }⊕ { x3•x2•x1, y3•y1, z1, w3•w1 }
      = { x3•x2•x1, z1 }
```

In effect, we are comparing the *specifications* of files to test for file equality, rather than the files themselves. The efficiency of doing so is significant: equational specifications are short (10s of bytes), and simple string matching is sufficient to test for equality. In contrast, the files that they represent are more expensive to generate and are long (1000s of bytes) where simple string matching is inefficient.

The algorithm, called *shared class extraction (SCE)*, finds the common equations in an arbitrary set of tools. It maps a set of *n* packages (one package per tool) to a set of *n+1* packages (one package per tool, plus the **Common** package):

$$\mathbf{SCE(\{T}_1\mathbf{, T}_2\mathbf{, \ldots T}_n\mathbf{\}) \rightarrow \{Common, T}_1\mathbf{', T}_2\mathbf{', \ldots T}_n\mathbf{'\}}$$

where:

$$\mathbf{Common = T}_1 \oplus \mathbf{T}_2 \oplus \mathbf{\ldots} \oplus \mathbf{T}_n$$

$$\mathbf{T}_1\mathbf{' = T}_1 - \mathbf{Common}$$
$$\mathbf{T}_2\mathbf{' = T}_2 - \mathbf{Common}$$
$$\mathbf{\ldots}$$
$$\mathbf{T}_n\mathbf{' = T}_n - \mathbf{Common}$$

and – is the set difference operator. Of course, there are variations of this algorithm. A file, for example, can be shared by some but not all of the input packages. For our study, we found the additional savings of these variants not worth the complexity.

---

1. Package names are implicit in AHEAD specifications.

The SCE optimization produces the same results as file diffing in terms of code archives. But SCE is more efficient because instead of diffing generated files, it identifies all common files by comparing file equations. In Section 3, we present our experimental results and a comparison between file diffing and our SCE optimization. It is worth noting that the SCE optimization does *not* rely on the fact that AHEAD tools are being built. The SCE algorithm imposes *no* interpretation on equations, which means it should be able to optimize the synthesis of *arbitrary* product-families in *arbitrary* domains. The same holds for our next optimization.

2.6 Merging Class Hierarchies (MCH)

A more sophisticated optimization relies on the knowledge that ATS tools are variants of a common design. Namely, all tools could be built using a common parser and a single AST class hierarchy. Each class of the hierarchy would have the form:

```
class typical extends ... {
   common methods and variables;
   jampack-specific methods and variables;
   mixin-specific methods and variables;
   ...
}
```

That is, each class would have a set of common methods and variables for traversing and editing ASTs, plus methods and variables that are specific to each tool. This might be the design of choice if ATS tools were developed manually. However, in typical FOP designs we generate a distinct class hierarchies for each tool by composing corresponding features. As a result common methods and variables are replicated in each class hierarchy, and consequently common code are shared by the generated tools. In this section, we show how the former design can be realized — and therefore the tool suite optimized — automatically using algebraic reasoning.

Consider the following general problem. Given tools $\mathbf{T}_1$, $\mathbf{T}_2$, …, $\mathbf{T}_n$, we want an operator $\otimes$ that *merges* their designs so that a single tool $\mathbf{T}_{n+1}$ has the union of the capabilities of each individual tool:

$$\mathbf{T}_{n+1} = \mathbf{T}_1 \otimes \mathbf{T}_2 \otimes \ldots \otimes \mathbf{T}_n \quad (5)$$

$\mathbf{T}_{n+1}$ has, in essence, all the features of all the tools that are merged. Like the common class extraction operator $\oplus$, the merge operator $\otimes$ is also associative and commutative:

$$(\mathbf{T1} \otimes \mathbf{T2}) \otimes \mathbf{T3} = \mathbf{T1} \otimes (\mathbf{T2} \otimes \mathbf{T3}) \quad (6)$$

$$\mathbf{T1} \otimes \mathbf{T2} = \mathbf{T2} \otimes \mathbf{T1} \quad (7)$$

Further, $\otimes$ distributes over encapsulation. That is, the merge of two tools is the same as the merge of its corresponding artifacts; tool-specific artifacts are just copied:

$$\begin{aligned}\mathbf{T3} \otimes \mathbf{T4} &= \{ \mathbf{a3}, \mathbf{b3}, \mathbf{c3} \} \otimes \{ \mathbf{b4}, \mathbf{c4}, \mathbf{d4} \} \\ &= \{ \mathbf{a3}, \mathbf{b3} \otimes \mathbf{b4}, \mathbf{c3} \otimes \mathbf{c4}, \mathbf{d4} \}\end{aligned} \quad (8)$$

That is, **(8)** is a special case of **(1)**.

Example. Consider the merge of the **jak2java** and **mixin** tools. The class hierarchies that are synthesized for **jak2java** and **mixin** are depicted in Figure 3a-b. A typical

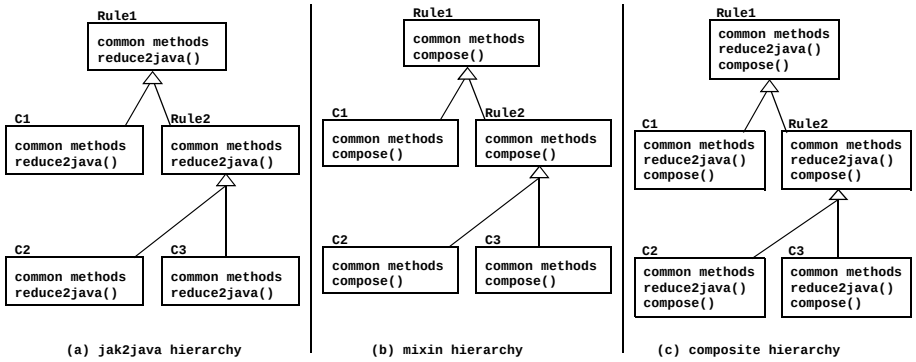

Figure 3. Tool-Specific and Merged Class Inheritance Hierarchies

class from **jak2java** has a set of common methods plus the **reduce2java()** method among others that translate or "reduce" an AST in Jak to an AST in Java. A typical class from the **mixin** tool has the same common methods. However, it has a **compose()** method among others that compose ASTs of different Jak files. The result of **jak2java** ⊗ **mixin** is a class hierarchy where each class has the union of the methods in the corresponding classes in each tool (Figure 3c). Merging is not limited to classes in the AST hierarchy; all classes in these tools participate. The parser classes, for example, do not belong to the AST hierarchy but are merged also. Since the parser classes in both tools are identical, they are simply copied to the merged tool.

In the following, we explain how class inheritance hierarchies are merged. We consider the same issues and make similar assumptions as Ernst [4] and Snelting and Tip [10], who studied the semantic issues of merging class inheritance hierarchies prior to our work.

2.6.1 Conflict-Freedom

Two classes can be merged into a single class if they agree on the contents (variables, methods, and constructors) that they share. Equivalently, *classes cannot be merged if they have different definitions for a shared variable, method, or constructor*. The property that two classes can be merged is called *conflict-freedom* or *static non-interference* [10].

In general, two AHEAD sets are *conflict free* if they agree on the specifications of the artifacts that they share. Consider sets $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$:

$\mathbf{A}_1 = \{ \mathbf{r}_1, \mathbf{s}_1 \}$
$\mathbf{A}_2 = \{ \mathbf{r}_1, \mathbf{t}_1 \}$
$\mathbf{A}_3 = \{ \mathbf{r}_2, \mathbf{t}_1 \}$

Sets $\mathbf{A}_1$ and $\mathbf{A}_2$ are conflict free because they share artifact **r** and the definitions for **r** are the same (both are $\mathbf{r}_1$). However, sets $\mathbf{A}_1$ and $\mathbf{A}_3$ conflict because they share artifact **r** and have different definitions for **r** ($\mathbf{r}_1$ is different than $\mathbf{r}_2$). Similarly, sets $\mathbf{A}_2$ and $\mathbf{A}_3$ conflict. They have **r** and **t** in common; the **t**'s are the same but their **r**'s differ.

We can automatically deduce if two classes are conflict free in the following way. Each tool to be merged is defined by a set of equations, one equation for each Jak class to synthesize. From a previous page, we defined an equation for class **w** for tool **T1** as:

```
w = w2 • w1                                                    (9)
```

And the corresponding equation for **w** in tool **T2** was:

```
w = w3 • w1                                                    (10)
```

Each w_i is a class or a class refinement that encapsulates a set of members. Suppose **w1**, **w2**, and **w3** are:

```
w1 = { a1, b1 }
w2 = { c2, b2 }
w3 = { d3, b3 }                                                (11)
```

That is, **w1** encapsulates members **a1** and **b1**; **w2** encapsulates **c2** and **b2**, etc.[1] We use **(11)** and the law of composition **(1)** to expand equations **(9)** and **(10)** to synthesize class specifications:

```
w = { a1, b2•b1, c2 }   // tool T1
w = { a1, b3•b1, d3 }   // tool T2
```

That is, class **w** for tool **T1** has data members or methods **a**, **b**, and **c**; class **w** for tool **T2** has data members or methods **a**, **b**, and **d**. Given these specifications, we see that they conflict — the two tools differ on their definitions for member **b**. This means the specifications of the **w** classes cannot be merged, and thus the **w** classes are placed in their tool-specific packages. If specifications can be merged, we merge them and place the merged class into a shared package.

Reflection. Java's reflection mechanism allows programs to do various kinds of self-inspection, e.g. to retrieve the name of an object's class and to determine the number of methods in a given class. Although reflection was not used in ATS tools, merged classes that use reflection may execute differently before and after a merge. There is no easy solution other than "rewriting existing code on a case by case basis as is deemed necessary" [14]. In our approach, a user can specify the classes that use reflection by listing them in a configuration file; these classes are not merged and are placed into tool-specific packages [12]. We discuss how to merge specifications in the next section.

2.6.2 The Merge Operator ⊗

Suppose the equations for class **y** in tools **T1** and **T2**, shown below, are expanded and are found not to conflict:

```
y = y2 • y1      // tool T1
y = y3 • y1      // tool T2
```

What is the merge **(y3•y1)** ⊗ **(y2•y1)** of these equations?

---

1. We do *not* compare the source code of method and data member definitions. b_i means the definition of member b in file i. We assume $b_i \neq b_j$ for all $i \neq j$.

The merge operator exploits the fact that its equations do not conflict and that it integrates equations by preserving the partial order relationships of individual equations. For example, the equations of **y** show that **y2** and **y3** are refinements of **y1**. Hence we have:

```
y2 > y1,
y3 > y1
```

where **>** indicates a partial order relationship between the class refinements. A merge of these partial orders yields another partial order.[1] The merge operator generates an equation that contains every term in its input and preserves the order imposed by each equation. If no order is specified for a particular pair of elements, then both permutations are legal. Thus, either of the following equations produce equivalent output:

```
(y2•y1) ⊗ (y3•y1)= y3•y2•y1
                 = y2•y3•y1
```

The correctness of a merge comes directly from its specification. Since there is no conflict, different refinements for a class in different tools are orthogonal — they do not affect each other and the order of their composition does not matter. So the merging of class specifications is correct as long as the compositional ordering in each equation is preserved. Thus, the result of **T1** ⊗ **T2** is:

```
Common  = T1 ⊗ T2
        = {x3•x2•x1, y2•y1, z1, w2•w1} ⊗ {x3•x2•x1, y3•y1, z1, w3•w1}
        = { (x3•x2•x1) ⊗ (x3•x2•x1), (y2•y1) ⊗ (y3•y1), z1 ⊗ z1,
            (w2•w1} ⊗ (w3•w1) }
        = { x3•x2•x1, y3•y2•y1, z1 }
```

For files that are not merged (like **w** above) because of conflicts, they do not appear in the merged result but do appear in the tool-specific packages **T1′** and **T2′**:

```
T1′ = { w2•w1 }
T2′ = { w3•w1 }
```

Thus, three packages **Common**, **T1′** and **T2′** are synthesized.

Once tools are merged, it remains to be shown that the merged code is type correct and that each merged tool has the same behavior as its unmerged counterpart [10]. We demonstrate type correctness and semantic equality of the merged tools in the appendix.

---

1. This is true for all ATS tools and for all tools that we can imagine. If a cycle were created by a merge, it would indicate either that the cycle could be eliminated by permuting features without changing tool semantics, or that there is a fundamental error in the design of the domain model. We have encountered the former which is easy to fix [2], but never the latter. In either case, the merge of the equations would fail, just as if the equations were recognized to be in conflict.

3 Experiments

We applied the two optimizations on the synthesis on the five ATS tools described earlier. Currently the size of these tools is 170K LOC. In our experiments, we used a desktop computer with an Intel Pentium III 733 Mhz microprocessor, 128 MB main memory running Microsoft Windows 2000 and Java SDK 1.4.1. Table 1 shows the number of classes, *lines of code (LOC)* and archive size of each ATS tool. The LOC measurement is calculated from the Java source code of each tool, and the archive size is obtained from the generated Java JAR files. In the original build without optimizations, a package is compiled for each tool.

Package	Classes	LOC(K)	Archive(KB)
jak2java	511	38	546
jampack	496	38	556
mixin	495	35	483
mmatrix	499	34	467
unmixin	496	34	457
total	2497	178	2,509

Table 1. Product-Family Statistics w/o Optimization

Table 2 demonstrates the results of shared class extraction optimization. The first five rows summarize each tool-specific package while the last row is the shared package. Nearly 70% of the classes in each tool are shared. Factoring these classes into a common package reduces the volume of code and executables by over 45%.

Package	Classes	LOC(K)	Archive(KB)
jak2java	165	13	246
jampack	150	13	257
mixin	149	10	189
mmatrix	153	9	170
unmixin	150	9	160
shared	347	25	294
total	1,114	80	1,316

Table 2. Product-Family Statistics of SCE Optimization

Table 3 lists the corresponding results for the merging class hierarchy optimization. All conflict-free classes are merged into the shared package, which leaves only conflicting classes in each tool-specific package. Conflicts in ATS tools are rare — of 500 classes in each tool, only 10 or 11 (including 5 dynamic interferences discussed in the appendix) conflict. This yields even greater reductions — more than 65% — in code and archive volume. Note that in Table 2 and Table 3, the number of classes in a tool-specific package plus the shared package is slightly larger than that of the original tool package shown in Table 1. For example, in Table 2 the total number of **`jak2java`** classes (165) and **`shared`** classes (347) is 512, whereas the original **`jak2java`** package has 511 classes as Table 1 shows. This is because some classes are not needed by all the tools, but they still can be factored out by SCE or merged by MCH optimization processes.

Package	Classes	LOC(K)	Archive(KB)
jak2java	10	2	32
jampack	11	3	46
mixin	11	3	35
mmatrix	10	2	25
unmixin	10	2	24
shared	510	43	689
total	562	55	851

Table 3. Product-Family Statistics of MCH

Table 4 illustrates the times of the unoptimized and optimized builds *that include overhead for optimizations*. It also shows the brute-force method to find common files by diffing generated files (**`Diff`**). In an unoptimized build, each class has to be composed from its featured source, compiled and finally packaged into jar files. Brute-force diffing reduces build times by 28%. SCE eliminates the need for unnecessary file generation and reduces build times by 39%. Comparing file specifications takes only three seconds since there is no file I/O. MCH has better performance as it reduces build times by 47%.

Build Time	Original	Diff	SCE	MCH
optimize	0	24	3	15
compose	170	170	134	110
compile	300	150	150	129
jar	26	14	14	7
total	496	358	301	261

Table 4. Build Time Comparisons

4 Related Work

Three topics are relevant to our work: composing class hierarchies, on-demand modularization, and AOP.

4.1 Composing Class Hierarchies

Refining a class hierarchy is equivalent to hierarchy composition. AHEAD, Hyper/J[7], and AspectJ[6] are among the few tools that can compose class hierarchies. Few papers address the semantic issues of hierarchy composition.

Snelting and Tip present algorithms for merging arbitrary class hierarchies [10]. Our work is a subproblem of what they addressed, and there are four basic differences. First, there is no known implementation of their algorithms [11]. Second, inheritance hierarchies that we merge are isomorphic by design. As mentioned earlier, the features that are composed in AHEAD have an implementation that conforms to a master design; this is how we achieve a practical form of interoperability and composability. Without pragmatic design constraints, features that are not designed to be composable won't be (or arbitrarily difficult problems may ensue). This is a variation of the architectural mismatch problem [5]. Third, the algorithm in [10] requires assumptions about the equivalence of methods in different hierarchies; we can deduce this information automatically from equational specifications. Thus our representations lead to more practical specifications of program relationships. Fourth, the means by which semantic equivalence is achieved in [10] requires verifying that each method call in the original and merged tools invoke the same method. Thus, if there are *n* tools, *c* is the number of classes per tool, *m* the number of methods per class, and *k* the number of calls per method, the cost of their algorithm to verify behavioral equivalence is $O(n*c*m*k)$. We achieve the same effect by comparing method signatures of each class to test for dynamic interference; our algorithm is faster $O(n*c*m)$ because it is more conservative.

Ernst considered a related problem of merging and reordering mixins [4]. Mixins approximate class refinements; the primary difference is that refinements can add and refine existing constructors, whereas mixins cannot. Ernst defines how mixins can be composed and how compositions of mixins can be merged. The technique of merging compositions is based on preserving partial orderings of compositions, just like our

work. However, the concept of composition is implicit in [4], and merging is the only explicit operator to "glue" mixins together. To us, composition and merge are very different operators that are *not* interchangeable — $(A \bullet B) \otimes (B \bullet C) \neq (A \bullet B) \bullet (B \bullet C)$. Thus, our model is more general.

4.2 On-Demand Remodularization

Ossher and Tarr were the first to recognize and motivate the need for *on-demand remodularization (ODM)*, which advocates the ability to translate between different modularizations [7]. While Hyper/J and AHEAD are tools that can be used for ODM, there are few published results or case studies on the topic.

Mezini and Ostermann proposed language constructs called *collaboration interfaces* to mix-and-match components dynamically [8]. The approach is object-based, where objects that fulfill contracts specified in collaboration interfaces are bound. The loose couplings of the implementations and interfaces allow collaborations to be reused independently. Here the purpose of remodularization is to meet the needs of different client programs, where in contrast we remodularize to optimize the program synthesis.

Lasagne [13] defines an architecture that starts with a minimal functional core, and selectively integrates extensions, which add new features to the system. A feature is implemented as a wrapper and can be composed incrementally at run-time. Dynamic remodularization is supported by the context sensitive selection on a per collaboration basis, enabling client specific customizations of systems. Our work also composes features, but it is done statically and AHEAD equations are algebraically optimized.

Our work remodularizes packages automatically by extracting common files into a shared package, thus eliminating redundancy and improving system build times. A similar result is described by Tip et al. [12], where Java packages are automatically optimized and compressed through the compaction of class inheritance hierarchies and the elimination of dead-code. Our work and [12] allows the user to specify where reflection occurs so that the corresponding classes may be properly handled to avoid errors. Our work is different because we split class inheritance hierarchies into multiple packages in order to optimize program achieve size and build time.

4.3 Aspect-Oriented Programming

AHEAD refinements have a long history, originating in collaboration-based designs and their implementations as mixins and mixin-layers (see [9] for relevant references). They also encapsulate *cross-cuts*, a concept that was popularized by *Aspect-Oriented Programming (AOP)* [6]. There are three differences between AOP and AHEAD. First, the concept of refinement in AHEAD (and its predecessor GenVoca) is virtually identical to that of extending object-oriented frameworks. Adding a feature to an OO framework requires certain methods and classes to be extended. AHEAD takes this idea to its logical conclusion: instead of having two different levels of abstraction (e.g., the abstract classes and their concrete class extensions), AHEAD allows arbitrary numbers of levels, where each level implements a particular feature or refinement [1].

Second, the starting points for AHEAD and AOP differ: product-lines are the consequence of pre-planned designs (so refinements are designed to be composable); this is

not a part of the standard AOP paradigm. Third, the novelty and power of AOP is in quantification. Quantification is the specification of where advice is to be inserted (or the locations at which refinements are applied). The use of quantification in AHEAD is no different than that used in traditional OO frameworks.

5 Conclusions

The synthesis of efficient software from declarative specifications is becoming increasingly important. The most successful example of this paradigm is relational query optimization (RQO). Replicating this paradigm in other domains and exploring its capabilities is the essence of our research.

In this paper, we focused on a key aspect of the RQO paradigm, namely the optimization of algebraic representations of programs. We showed how algebraic representations of the tools of a product-family could be automatically remodularized (refactored) so their shared infrastructure need not be replicated. We presented two optimizations that remodularized synthesized tool packages: extracting shared files and merging class hierarchies. Our optimizations are examples of equational reasoning; they were defined algebraically, were automatic, and required minimal domain knowledge. Further, our optimizations were efficient and practical: in both cases, we improved upon algorithms that previously existed. We presented a case study of a product-family of five tools and achieved a reduction of 40% in build times and archive size.

We believe our results contribute further evidence that algebraic representations of programs coupled with algebraic reasoning is a powerful way to express software designs and manipulate them automatically.

Acknowledgements. We thank Jacob Sarvela, Kurt Stirewalt, William Cook, and Mark Grechanik for their helpful comments on earlier drafts of this paper.

6 References

[1] D. Batory, J.N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement", *IEEE Transactions on Software Engineering*, June 2004.

[2] D. Batory, J. Liu, J.N. Sarvela, “Refinements and Multi-Dimensional Separation of Concerns”, *ACM SIGSOFT 2003 (ESEC/FSE2003)*.

[3] D. Batory, "The Road to Utopia: A Future for Generative Programming". Keynote presentation at *Dagstuhl for Domain-Specific Program Generation*, March 23-28, 2003.

[4] E. Ernst, “Propagating Class and Method Combination”, *ECOOP 1999*.

[5] D. Garlan, R. Allen, and J. Ockerbloom, “Architectural Mismatch: Why it is hard to Build Systems from Existing Parts”, *ICSE 1995*.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kirsten, J. Palm, W.G. Griswold. “An overview of AspectJ”, *ECOOP 2001*.

[7] H. Ossher and P. Tarr, “On the Need for On-Demand Remodularization”, Position Paper for Aspects and Dimensions of Concern Workshop, *ECOOP 2000*.

[8] M. Mezini and K. Ostermann. “Integrating independent components with on-demand remodularization”, In *Proceedings of OOPSLA '02*, 2002.

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

[10] G. Snelting and F. Tip, “Semantics-Based Composition of Class Hierarchies”, *ECOOP 2002*, 562-584.

[11] G. Snelting, personal communication.

[12] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. “Practical experience with an application extractor for Java”, In *Proceedings of OOPSLA*, pages 292--305, November 1999.

[13] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jrgensen. “Dynamic and Selective Combination of Extensions in Component-Based Applications”. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, Toronto, Canada, May 2001.

[14] O. Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. Ph.D. thesis, Stanford University, 1995.

Appendix

This appendix demonstrates type correctness and semantic equality of the merged tools. Static type correctness is simple. All ATS tools are variants of a master design. A design defines a set of class hierarchies; AHEAD refinements add more members to existing classes or add new classes only as bottom-level leaves to pre-defined hierarchies. Changing superclass relationships in inheritance hierarchies is not permitted, nor is deleting classes. Thus, the inheritance hierarchies that are present in a tool prior to merging remain the same after merging. Similarly, since methods are never deleted, the set of method *signatures* that are present in a class prior to merging are present afterwards. Thus, all objects created in an unmerged tool will be of the same type as that in the merged tool; all methods in the unmerged tool are present in the merged tool. If the unmerged tool is type correct, its corresponding code in the merged tool is type correct.

Proving behavioral equivalence between the unmerged and merged tools is more difficult. Although the general problem is undecidable, Snelting and Tip [10] have shown for merging class hierarchies, behavior equivalence can be checked via static analysis of *dynamic interference*. To verify that two tools (before and after merging) do not have dynamic interference, [10] requires us to show that (a) both define methods in the same way and (b) both invoke the same methods in the same order.

Same Method Definitions. The only problematic scenario is that in an unmerged program a class inherits a method from its superclass, but after merging this method is overridden. Figure 4a illustrates a class hierarchy before merging, where the method **`foo()`** is inherited by class **`Two`**. After merging, a different version of **`foo()`** is inserted in class **`Two`** that overrides the inherited method.

```
class One {
   void foo() {...}
}

class Two extends One {
  void main() {
    Two t = new Two();
    t.foo();
  }
}
```

(a)

```
class One {
   void foo() {...}
}

class Two extends One {
  void main() {
    Two t = new Two();
    t.foo();
  }
  void foo() {...}
}
```

(b)

Figure 4. Method Overridden in a Class Composition

A variant of (1) allows us to propagate the contents of class ancestors to its subclasses. Figure 5a shows a hierarchy of three classes and the members that they locally encapsulate. Figure 5b shows the contents of class encapsulation after propagation. Note that a method refinement (Δm_1 for example) extends the original method m_1 by performing some task intermixed with a *super* call.

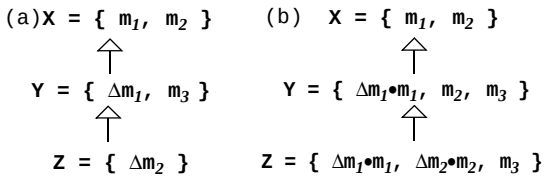

Figure 5. Propagating Contents Down A Hierarchy

Given this, we can determine the variables and methods of every class in each tool and the merged tool, along with their specifications by tracing back along the inheritance chains. Let C_i denote a class from tool T_i and C_m denote the corresponding class in the merged tool. If C_i does not conflict with C_m, we know C_m includes the same variables and methods of C_i *and* defines them in the same way. By performing this test over all classes in all original tools, we can prove that all methods in the original tools are present and are defined in the same way as in the merged tool.

Same Methods Called. We still need to prove that the same methods are called. Consider the class hierarchy of Figure 6a. When the **main** method is executed, the **foo(One x)** method is invoked. Now consider the addition of a specialized **foo(Two x)** method in Figure 6b. When **main** is now executed, **foo(Two x)** is called. Here is an example where all of the original methods in Figure 6a are present in Figure 6b, but at run-time a different, more specialized method is invoked, thus leading to different behavior. This is the problem of ambiguous method invocations.

(a)
```
class One {
  void foo(One x) {...}
}

class Two extends One {
  void main() {
    Two t = new Two();
    t.foo(t);
  }
}
```

(b)
```
class One {
  void foo(One x) {...}
}

class Two extends One {
  void main() {
    Two t = new Two();
    t.foo(t);
  }
  void foo(Two x) {...}
}
```

Figure 6. The Problem of Method Specialization

To detect this problem, we again return to the members that we computed for class C_i and C_m. Although we have been using simple names, like "m_1", to denote a class member, the actual name of a member is its type signature. By comparing type signatures of two methods we can tell whether one method is a specialization of another. If there is any method in the set difference $C_m - C_i$ (i.e., the methods in the merged class that are not members of the original class) that could be a specialization of a method in C_i, ambiguous invocation as in Figure 6 is possible. In our optimization process, potential ambiguous invocations are detected, and the corresponding classes are not merged and are put in tool-specific packages.

So assuring that all methods in the original tools are present and are defined in the same way as in the merged tool, and ambiguous method invocations are not possible, we guarantee the absence of dynamic interference, thus behavior equivalence between the unmerged and merged tools.