Feature Refactoring a Multi-Representation Program into a Product Line

Salvador Trujillo
Department of Computer Sciences
University of the Basque Country
20009 San Sebastian, Spain
struji@ehu.es

Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
batory@cs.utexas.edu

Oscar Diaz
Department of Computer Sciences
University of the Basque Country
20009 San Sebastian, Spain
oscar.diaz@ehu.es

Abstract

Feature refactoring is the process of decomposing a program into a set of modules, called *features*, that encapsulate increments in program functionality. Different compositions of features yield different programs. As programs are defined using multiple representations, such as code, makefiles, and documentation, feature refactoring requires *all* representations to be factored. Thus, composing features produces consistent representations of code, makefiles, documentation, etc. for a target program. We present a case study of feature refactoring a substantial tool suite that uses multiple representations. We describe the key technical problems encountered, and sketch the tool support needed for simplifying such refactorings in the future.

Categories and Subject Descriptors: D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, Reverse engineering, and Reengineering; D.2.13 [**Reusable Software**]: Domain engineering; I.2.2 [**Automatic Programming**]: Program synthesis.

***General Terms*:** Design, Experimentation.

***Keywords*:** Refactoring, software product lines, multiple representations, refinements, AHEAD, feature-oriented programming, program synthesis.

1 Introduction

Features are increments in program functionality. They are the semantic units by which different programs within a family or *software product line (SPL)* can be differentiated and defined. Different compositions of features yields different programs within an SPL.

An SPL often starts with a single program [12]. Variants of the program arise, and instead of maintaining multiple distinct versions, it is more cost effective to maintain the features from which different versions can be assembled. *Feature refactoring* is the process of decomposing a program into a set of features. Given such a refactoring, variants of the original program can be created by omitting unnecessary features. Further, new features can be added, and yet more programs can be assembled. This is how an SPL emerges. Program evolution is often enhanced by feature decompositions, as programs typically evolve by updating, adding, and deleting increments of functionality (features) [6][22][28].

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.

A program has many representations. Beside source code, there may also be regression tests, documentation, makefiles, UML models, performance models, etc. When a program is refactored into features, *all* of its representations — code, regression tests, documentation, etc. — must be refactored as well. That is, a feature encapsulates all representations (or changes to existing representations) that define the feature's implementation. When a program is built by composing features, all relevant representations are synthesized.

In this paper, we present a case study in feature refactoring that demonstrates these concepts. The program that we refactor is the *AHEAD Tool Suite (***ATS***)* which is a collection of tools that were developed for feature-based program synthesis [1]. Over time, **ATS** has grown to 24 different tools expressed in over 200K LOC Java. In addition to code, there are makefiles, regression tests, documentation, and program specifications, all of which are intimately intertwined into an integrated whole. There has been an increasing need to customize **ATS** by removing or replacing certain tools. This motivated the feature refactoring of **ATS**, in order to create a product line of its variants.

What is new about our work is the scale of refactoring. Prior work on feature refactoring dealt with small programs under 5K LOC and focussed only on code refactoring [21][22]. In this case study, we scale features substantially in size (**ATS** is almost two orders of magnitude larger) and in the kinds of representation (**ATS** required refactoring not only code, but documentation, makefiles, and regression tests as well). This wholistic approach to refactoring *information* is fundamental to feature orientation, and as such this paper constitutes a valuable case study on the scalability of feature-based program refactoring and synthesis.

We describe the technical problems encountered in feature refactoring **ATS**, and the kinds of tool support needed for performing such refactorings in the future. We believe our experiences (except one) extrapolate to the feature refactoring of other tool suites. The exception is that **ATS** is bootstrapped, and it posed its own special conceptual and technical challenges. We begin with a description of AHEAD and **ATS**.

2 Feature Oriented Programming

Feature Oriented Programming (FOP) is a general paradigm of program synthesis. Features (a.k.a. feature modules) are the building blocks of programs. Each feature may include any number of *artifacts* (i.e., representations). GenVoca was an early model of FOP; AHEAD is the current model.

2.1 GenVoca

A GenVoca model of an SPL is an algebra that offers a set of operations, where each operation implements a feature. We write `M = {f, h, i, j}` to mean model `M` has operations or features `f`, `h`, `i`, and `j`. GenVoca distinguishes features as *constants* or *functions*. Constants represent base programs. For example:

```
f       // a program with feature f
h       // a program with feature h
```

Functions represent *program refinements* that extend a program that is received as input. For instance:

```
i•x     // adds feature i to program x
j•x     // adds feature j to program x
```

where • denotes function application.

The design of a program is a named expression, e.g.:

```
prog1 = i•f     // prog1 has features f and i
prog2 = j•h     // prog2 has features h and j
prog3 = i•j•h   // prog3 has features h,j,i
```

The set of programs that can be created from a model is its product line. Expression optimization corresponds to program design optimization, and expression evaluation corresponds to program synthesis [27][3][20]. Not all features are compatible. The use of one feature may preclude the use of some features or may demand the use of others. Tools that validate compositions of features are discussed in [7].

2.2 AHEAD

Algebraic Hierarchical Equations for Application Design (AHEAD) extends GenVoca to express nested hierarchies of *artifacts* (i.e., files) and their composition [6]. If feature `f` encapsulates a set of artifacts a_f, b_f, and d_f we write `f = {` a_f, b_f, d_f `}`. Similarly, `i = {` a_i, b_i, c_i `}` says that feature `i` encapsulates artifacts a_i, b_i, and c_i. As artifacts themselves may be sets, a feature is a nested set of artifacts. AHEAD uses directories to represent nested sets. Figure 1a shows an AHEAD feature and its corresponding directory.

The composition of features is governed by the rules of inheritance. In the composition `i•f`, all artifacts of `f` are inherited by `i`. Further, artifacts with the same name (ignoring subscripts) are composed pairwise. This is AHEAD's *Law of Composition*:

$$\begin{aligned} i•f &= \{ a_i, b_i, c_i \} • \{ a_f, b_f, d_f \} \\ &= \{ a_i•a_f,\ b_i•b_f,\ c_i,\ d_f \} \end{aligned} \quad (1)$$

Features are composed by applying `(1)` recursively, where directories are folded together by composing corresponding artifacts in each directory. Figure 1b shows the composition of features `A` and `B`. The result is feature `C`, where artifact `X.jak` of `C` is synthesized by composing `X.jak` (from `B`) with `X.jak` (from `A`) [6].

The polymorphism of the • operator is central to AHEAD. Artifacts of a given type (`.jak`, `.b`, etc.) and their refinements are defined in a type-specific language. That is, the definition and refinements of `.jak` files are expressed in the Jak(arta) language, a superset of Java. The definition and refinement of `.b` files are expressed as Bali grammars, which are annotated BNF files. And so on. One or more tools implement the • operator for each artifact type. The `jampack` and `mixin` tools implement the • operator for `.jak` files (i.e., `.jak` files are composed by either the `jampack` and `mixin` tools), and the `balicomposer` tool implements the • operator for `.b` files. `ATS` is the set of tools that compose artifacts, produce and analyze compositions, and derive artifacts of one type from others (e.g., the `jak2java` tool translates a `.jak` file to its `.java` counterpart).

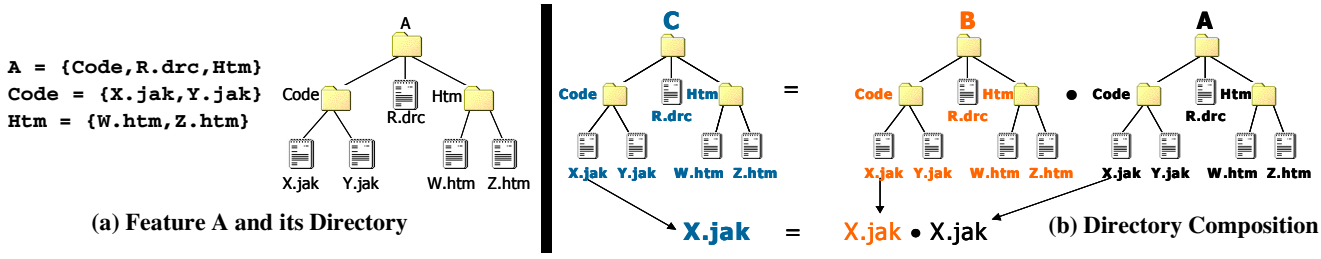
Notation. Although we write the composition of `A` and `B` as `A•B`, it really is an abbreviation of the expression `compose(A,B)`, where `A` and `B` denote feature directories. We use • to simplify our expressions.

2.3 XAK and XML Document Refinement

`XAK` (pronounced "sack") is a tool for composing base and refinement artifacts in XML format. The need for `XAK` arose two years ago while developing SPLs for web applications using AHEAD [16]. An unusual characteristic of web applications is that a sizable fraction of their definition is not Java or Jak source, but rather XML specifications [16] (e.g., JSP, HTML, and Struts control flow files [2]). Thus, it is common for a feature of a web application to contain XML artifacts (base or refinements) and Jak source (base or refinements). We began our work at a time when AHEAD did not have a language for XML artifact refinement and a tool for XML artifact composition. This lead to the creation of `XAK`.

`XAK` follows the AHEAD paradigm of module definition and refinement. An XML document is a tree rooted at node `t1` in Figure 2a. Its `XAK` counterpart is slightly different: it is a tree of trees (e.g., trees rooted at nodes `t1`, `t2`, and `t3` in Figure 2b), and each of these nodes is tagged with a `xak:module` attribute, so that the `XAK` module abstraction of the original tree is a tree of modules (module `m1` contains modules `m2` and `m3` in Figure 2c).

In general, a `XAK` module has a unique name and contains one or more consecutive subtrees that share the root module. Each subtree may contain any number of modules. Note that the modules of an XML artifact reflect a natural hierarchical partitioning into semantic units that can be refined (more on this shortly).



(a) Feature A and its Directory

(b) Directory Composition

Figure 1. Features as Directories

As a concrete example, Figure 3a shows a **XAK** artifact that defines a bibliography. The artifact is partitioned into modules (see the **xak:module** attributes) which impose the module structure of Figure 3b. Also, the root of the artifact is labeled with the **xak:artifact** attribute. Note that all modules have unique names.

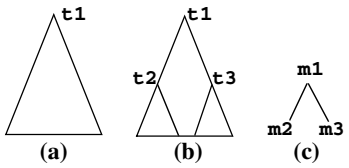


Figure 2. XAK Module Hierarchy

```
<bibliography xak:artifact="myBib">
    <paper xak:module="mATSRefac">
        <title>Feature Refactoring ..</title>
        <date>06-02-21 11:30:45</date>
        <authors xak:module="mATSAuthors">
            <author id="1">
                <name>Oscar</name>
            </author>
        </authors>
    </paper>
    <paper xak:module="mXAKRef">
        <!-- content collapsed -->
    </paper>
</bibliography>
```

(a)

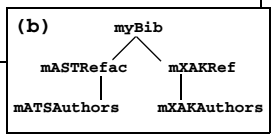


Figure 3. XAK Base doc

A refinement of a **XAK** module is defined similarly to method refinement in the Jak language [6]. A refinement of the **XAK** artifact of Figure 3 is shown in Figure 4 that appends a new author to the **mATSAuthors** module. The **xak:super** node is a marker that indicates the place where the original module body is to be substituted. In general, a **XAK** refinement artifact can contain any number of **xak:module** refinements.

```
<xak:refines xak:artifact="myBib">
    <xak:extends xak:module="mATSAuthors">
        <xak:super xak:module="mATSAuthors"/>
        <author id="2">
            <name>Don</name>
        </author>
    </xak:extends>
</xak:refines>
```

Figure 4. XAK Refinement

XAK is a tool that composes a base **XAK** artifact with zero or more refinements. (**XAK** also composes refinements into a compound refinement). The result of composing the base artifact of Figure 3 and the refinement of Figure 4 is the artifact in Figure 5. Note that it is possible for a **XAK** refinement to add new modules. In this example, only new content is added to an existing module. Thus, the modular structure of Figure 3b is unchanged.

The result of a **XAK** composition is a **XAK** artifact. The underlying XML artifact is the **XAK** artifact with the **xak:artifact** and **xak:module** attributes removed. **XAK** has other functionalities not needed for our work, such as interfaces, schema extensions and validation [17].

```
<bibliography xak:artifact="myBib">
    <paper xak:module="mATSRefactor">
        <title>Feature Refactoring ..</title>
        <date>06-02-21 11:30:45</date>
        <authors xak:module="mATSAuthors">
            <author id="1">
                <name>Oscar</name>
            </author>
            <author id="2">
                <name>Don</name>
            </author>
        </authors>
    </paper>
    <!-- content collapsed -->
</bibliography>
```

Figure 5. XAK Composition Result

2.4 Feature Refactoring and ATS

Feature refactoring is the inverse of feature composition. Instead of starting with a base program **B** and features **F** and **G**, and composing them to build program **P = F•G•B**, feature refactoring starts with **P** and refactors **P** into an expression **F•G•B** [22].

Figure 6 shows a part of the directory structure of **ATS**. There is an expression per tool in the **ahead/expressions** directory. **ahead/lib** contains *Java archive (JAR)* files for **ATS**, **ahead/models** contains directories of AHEAD models, **fopdocs** contains HTML tool documents, and **regression** is a directory of regression tests.

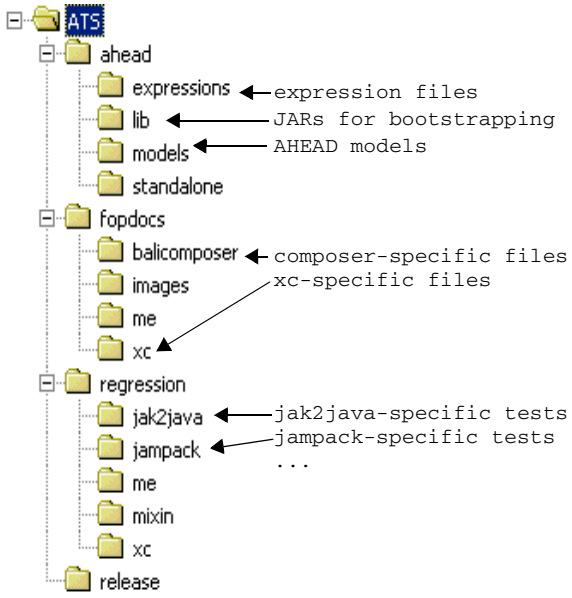


Figure 6. The ATS Directory Structure

ATS is the baseline program for a product line of variants. We want to feature refactor **ATS** into a **core** (the kernel of **ATS**) and optional features, one per tool (e.g., **aj**, **cpp**, **drc**, **jedi**, etc.). By doing so, we create an AHEAD model of **ATS**, which we call the *ATS Product Line* (**APL**):

```
APL = { core,   // kernel of ATS
        aj,     // aspectj translator
        cpp,    // c++ tools
        guidsl, // feature modeling tool
        xc,     // XML composer
        drc,    // design rule tool
        jedi,   // javadoc-like tool
        me,     // modelexplorer
        ... }
```

Given **APL**, we can synthesize different variants of **ATS**:

```
ATS1 = ...cpp•drc•xc•core;  // full set of tools
ATS2 = ...cpp•core;         // a subset of tools
ATS3 = jedi•guidsl•me•core
ATS4 = drc•aj•core
```

An essential tool in synthesizing variants of **ATS** is **XAK**, whose capabilities we described in the previous section.

3 The Process of Feature Refactoring ATS

Let $\mathbf{ATS}_{\mathbf{src}}$ denote the AHEAD feature that contains the source artifacts for the AHEAD Tool Suite. The tool binaries (i.e., JAR files) are produced by an **ant** XML build which we model by the function **antBuild**:

$$\mathbf{ATS}_{\mathbf{bin}} = \mathbf{antBuild}\ (\mathbf{ATS}_{\mathbf{src}})$$

That is, $\mathbf{ATS}_{\mathbf{bin}}$ differs from $\mathbf{ATS}_{\mathbf{src}}$ in that tool binaries have been created and added to the **ATS** directory. The build process itself creates directories to contain tool JARs, newly created batch and bash executables [13], and runs the regression tests to evaluate the correctness of tool executables. An **ATS** build time is about one-half hour.

Figure 7 illustrates the five-step process that we used to feature refactor $\mathbf{ATS}_{\mathbf{src}}$ into an SPL of **ATS** variants.

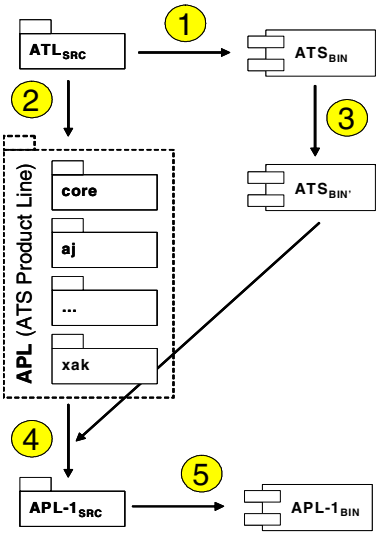


Figure 7. ATS Refactoring Process

Step 1. $\mathbf{ATS}_{\mathbf{bin}}$ was created by an **ant** build of $\mathbf{ATS}_{\mathbf{src}}$:

$$\mathbf{ATS}_{\mathbf{bin}} = \mathbf{antBuild}\ (\mathbf{ATS}_{\mathbf{src}})$$

Step 2. $\mathbf{ATS}_{\mathbf{src}}$ was feature refactored (more on this in Section 3.1) into the **APL**:

```
APL = { core, jedi, cpp, aj ...}
```

Each **APL** feature has **XAK** artifacts, that either define or refine HTML and XML documents in $\mathbf{ATS}_{\mathbf{src}}$.

Step 3. Although **APL** is itself an AHEAD model, we could *not* compose its features using the binaries of $\mathbf{ATS}_{\mathbf{bin}}$ created in Step 1. The reason is that **APL** features encapsulate **XAK** artifacts and their refinements, and the **XAK** tool is *not* part of $\mathbf{ATS}_{\mathbf{bin}}$. However, **XAK** *can* be added to $\mathbf{ATS}_{\mathbf{bin}}$ by refinement. Let $\mathbf{ATS}_{\mathbf{xak}}$ be a feature that adds the **XAK** tool to $\mathbf{ATS}_{\mathbf{bin}}$ to produce $\mathbf{ATS}_{\mathbf{bin'}}$, which is a set of tools that *can* compose **APL** features:

$$\mathbf{ATS}_{\mathbf{bin'}} = \mathbf{ATS}_{\mathbf{xak}} \bullet \mathbf{ATS}_{\mathbf{bin}} \qquad (2)$$

Note that the tools of $\mathbf{ATS}_{\mathbf{bin}}$ are used to evaluate **(2)**. As we explain in Section 3.2, this is a form of bootstrapping where a tool suite is used to build a new version of itself.

Step 4. We use the tools of $\mathbf{ATS}_{\mathbf{bin'}}$ to synthesize different variants of $\mathbf{ATS}_{\mathbf{src}}$ by composing **APL** features, such as:

$$\mathbf{APL\text{-}1}_{\mathbf{src}} = \mathbf{aj} \bullet \mathbf{core}$$

$$\mathbf{APL\text{-}2}_{\mathbf{src}} = \mathbf{jedi} \bullet \mathbf{core}$$

Step 5. Tool binaries of an **ATS** variant are obtained by an **ant** build:

$$\mathbf{APL\text{-}1}_{\mathbf{bin}} = \mathbf{antBuild}\ (\mathbf{APL\text{-}1}_{\mathbf{src}})$$

$$\mathbf{APL\text{-}2}_{\mathbf{bin}} = \mathbf{antBuild}\ (\mathbf{APL\text{-}2}_{\mathbf{src}})$$

In the following sections, we elaborate the details of steps 2, 3, and 4 in this process.

3.1 Step 2: Refactoring ATS

One of the core contributions of SPL research is the use of features and feature models to define members of an SPL. Although the process of identifying features, refactoring them from the original program, and creating a feature model is iterative, we proceeded in a largely straightforward fashion (discussed below) with minimal backtracking.

3.1.1 The APL Feature Model

We identified the following features in **ATS**:

- **core**. The kernel of **ATS** includes Jak file tools **jampack**, **mixin**, and **jak2java**, and Bali file tools such as **balicomposer** and **bali2jak**.
- **aj**, **mmatrix**, **jedi**, **reform**. Tools that process Jak files. **aj** translates Jak files to AspectJ files, **mmatrix** collects statistics on Jak files, **jedi** is a javadoc-like tool, and **reform** is a Jak file pretty-printer.

- **guidsl**, **web**, **me**. GUI-based tools for declaratively specifying programs and exploring AHEAD models.
- **xak**, **xc**. Tools for composing XML files.
- **bctools**. Tools that produce, compose, and analyze byte codes.
- **obe**, **drc**, **cpp**. Miscellaneous tools.

Figure 8 depicts a feature model for **APL** using a notation similar to [14].

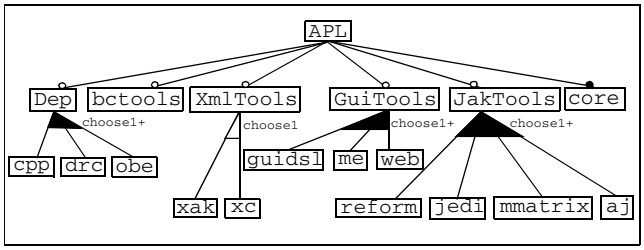


Figure 8. APL Feature Model

Figure 9 shows an alternative grammar representation that is used in AHEAD to specify feature models and cross-tree constraints (e.g., the selection of **bctools** or the **me** tool requires the **mmatrix** tool) [7]. Note that the cross-tree constraints were discovered during the refactoring of **ATS**, discussed in the next section.

```
// APL Grammar

APL : Deprecated* [cpp] [bctools] [XmlTools]
      GuiTools* JakTools* core ;

JakTools : // optional Jakarta tools
           reform | jedi | mmatrix | aj ;

GuiTools : guidsl | web | me ; // GUI-based tools

XmlTools : xc | xak ; // XML composition tools

Deprecated: obe | drc | cpp ; // miscellaneous

%% // constraints

bctools ∨ me ⇒ mmatrix;
```

Figure 9. APL Grammar

3.1.2 The Refactoring Process

We refactored $\mathbf{ATS_{src}}$ in a progressive manner. First, we identified the kernel of $\mathbf{ATS_{src}}$ which we called the **core** feature. The remainder of $\mathbf{ATS_{src}}$ was called $\mathbf{extra_0}$. So our first step in refactoring **ATS** was described by:

$$\mathbf{ATS_{src} = extra_0 \bullet core} \quad \mathbf{(3)}$$

We knew that we found a correct definition of **core** when we were able to build **core** tools and run their regression tests without errors:

$$\mathbf{ATS_{core} = antBuild\ (core)} \quad \mathbf{(4)}$$

Unlike prior feature refactoring work, the presence of regression tests in a feature helped us confirm that our refactoring was correct. If a build failed, we tracked down the missing pieces as files that had to be moved from $\mathbf{extra_0}$ into **core** — i.e., we failed to include all parts of **core** in our refactoring of **(3)**.

Next, we factored out feature $\mathbf{F_n}$ from $\mathbf{extra_n}$ to produce a smaller $\mathbf{extra_{n+1}}$ feature:

$$\mathbf{extra_n = extra_{n+1} \bullet F_n} \quad \mathbf{(5)}$$

To verify that we had a correct definition of $\mathbf{F_n}$, we composed it with **core**, built the binaries, and ran the regression tests correctly, which takes over 20 minutes:

$$\mathbf{ATS_n = antBuild\ (F_n \bullet core)} \quad \mathbf{(6)}$$

Again, the regression tests validated both the **core** tools and the additional tool(s) encapsulated in $\mathbf{F_n}$. Failed builds helped us identify pieces that were not moved from $\mathbf{extra_{n+1}}$ to $\mathbf{F_n}$.

We repeated **(5)** and **(6)** to remove incrementally all features from **extra** that we identified in the last section. The final version of **extra** was empty — it contained no files. This refactoring took around 10 person/days.

This process required adjustment when we discovered dependencies among **APL** features. For example, the **bctools** feature required the **mmatrix** feature because the **bctools** invoked the **mmatrix** tool. We had to create a version **v** of **ATS** with both tools before we could build and test:

$$\mathbf{ATS_v = antBuild(\ bctools \bullet mmatrix \bullet core\ )}$$

3.1.3 Feature Contents

Each feature encapsulated a tool or group of tools that roughly contained the same content. There were new artifacts such as source files, makefiles, HTML documentation, and regression tests specific to that feature. Refinements were generally limited to **ant** makefiles and HTML document files, both of which were encoded as **XAK** files and **XAK** refinements.

The refactorings of **(3)** and **(5)** were straightforward. We first identified the parts of **ant** makefiles that triggered the building of a target tool **T**, its parts, and its regression tests. These statements were factored from **extra** into a **XAK** file that refined the **ant** build script of **core**. (Initially, we simply commented them out, and once we knew we had a correct set of statements, we moved them into a **XAK** refinement file). We followed a similar procedure for refactoring HTML documents. The remaining work was to move files (source, HTML, regression) that were specific to the tools of **T** into the feature **T**, being careful to retain the directory hierarchy in which these files were to appear. There were other files in addition to **ant** makefiles and HTML documents that had to be refined, but these were few and simple (e.g., largely text file concatenation); AHEAD had tools to perform these refinements.

Figure 10 shows the detail of disk volume, the number of files and their number of lines of code in each **ATS** feature. The **FILES** column indicates the total number of files in each feature. The **Java**, **Jak**, and **XML** files were introductions (constants), while **XAK** files indicates the number of files that an **ATS** feature refines.

	SIZE	# FILES	#Java	#LOC	#Jak	#LOC	#XML	#LOC	#XAK	#LOC
core	33M	3904	1072	162878	1326	65243	78	24830	0	0
aj	106K	28	0	0	25	1293	2	41	2	41
bc	903K	88	13	3406	44	1285	13	2140	5	81
cpp	128K	48	4	501	0	0	8	1241	5	103
drc	348K	149	2	93	24	1708	7	1128	5	80
guidsl	901K	255	0	0	225	11631	5	1186	3	50
jedi	884K	200	0	0	133	12250	5	740	3	53
jrename	55K	28	9	406	5	88	2	29	2	29
me	7.7M	393	83	17862	96	2583	7	3237	3	53
mmatrix	34K	8	0	0	0	0	6	412	5	84
obe	495K	41	23	2369	0	0	10	1072	7	128
reform	712K	114	0	0	105	2303	6	787	4	60
xak	3.9M	56	17	1736	0	0	20	2757	9	171
xc	1.4M	42	5	1002	0	0	21	4896	4	114

Figure 10. Feature Size and Content Distribution

3.2 Step 3: Bootstrapping `ATS/lib`

One of the unique parts about refactoring **ATS** is its reliance on bootstrapping: to build **ATS** tools requires **ATS** tools. Many **ATS** tools are written in the Jak language and are themselves composed from features. Seven **ATS** tools are needed for bootstrapping:

- **jampack**, **mixin** — tools to compose Jak files,
- **jak2java** — a Jak to Java file translator,
- **balicomposer** — a tool to compose Bali files.
- **bali2jak** — a Bali file to Jak file translator,
- **bali2javacc** — a Bali file to **JavaCC** translator, and
- **composer** — a tool that composers features.

These seven tools are stored as JAR files, which we call *bootstrapping JARs*, in directory **ATS/lib** (see Figure 6). As soon as one of the above tools is synthesized during an **ATS** build, its JAR file replaces its bootstrapping JAR from that point on in an **ATS** build. At the end of an **ATS** build, no bootstrapping JARs are used.

To synthesize customized versions of **ATS** from features required three distinct changes to be made to **ATS** itself. What makes this intellectually challenging is whether the changes could be expressed using AHEAD concepts. We wanted to stress AHEAD to understand better its generality. In the following, we outline the three changes that we made, and explain how we realized them using refinements.

First, most **APL** features encapsulated regression tests that were specific to the tool(s) the feature encapsulated. Among the tests for Jak tools are Jak files that are syntactically incorrect. (These files are used to test language parsers). The Jak file composition tools (**mixin** and **jampack**) parse all Jak input files before performing any actions. We refined both tools so that if only one Jak file was listed on their command line, the file itself would not be parsed, but merely copied to the target feature directory. In this way, **composer** could invoke **mixin** or **jampack** on syntactically incorrect Jak files without discovering their syntactic errors.

Second, we needed to add **xak.jar** to the bootstrapping JARs to compose **XAK** files representing HTML documents and **ant** build scripts.

Third, we needed to refine JAR files. In particular, the bootstrapping JAR file for **composer**. Remember that this JAR contains a version of **composer** that understands how to compose a predefined set of artifacts (Jak files, Bali files, etc.). If **composer** is to compose new artifact types (such as **XAK** files), **composer**'s bootstrapping JAR must be refined to add this capability *before* it can compose **APL** features.

composer was designed so that new artifact-composing tools could be added easily. For each artifact type, two Java class files are placed in a subdirectory (called **Unit**) of the **composer** tool. These class files contain information that tell **composer** how to invoke a tool to compose artifacts of a specific type. For example, there is a pair of files that tell **composer** how to compose Jak files using the **mixin** tool; there is another pair of files that tells **composer** how to compose Bali grammar files using the **balicomposer** tool, etc. To support **XAK** file composition, two additional class files had to be added to the **Unit** directory to tell **composer** how to compose **XAK** files.

All three changes could be made by refining **ATS/lib** using **jarcomposer**. We chose to modify **mixin** and **jampack** directly as its changes were permanent, while adding **xak.jar** and its modification of **composer.jar** in **ATS/lib** were optional. We accomplished the last two changes (adding **xak.jar** and refining **composer.jar**) by refining **ATS/lib**. The general problem is as follows: **lib** is itself an AHEAD module that contains bootstrapping JARs:

```
lib = { composer.jar, mixin.jar, ... }
```

A refinement of **lib** includes new JAR files plus a refinement of the **composer.jar** to tell **composer** how to compose new file types using the new JAR files. For example, the refinement to **lib** that adds **XAK** is:

$$\mathbf{lib_{xak}} = \{\ \mathbf{composer.jar_{xak}},\ \mathbf{xak.jar}\ \}$$

where $\mathbf{composer.jar_{xak}}$ contains the **Unit** file extensions of **composer** that calls **XAK**. To produce a refined **lib** (denoted $\mathbf{lib_{new}}$ below), we compose $\mathbf{lib_{xak}}$ with **lib**:

$$\begin{aligned}\mathbf{lib_{new}} &= \mathbf{lib_{xak}} \bullet \mathbf{lib} \\ &= \{\ \mathbf{composer.jar_{xak}} \bullet \mathbf{composer.jar},\\ &\quad\ \mathbf{xak.jar},\ \mathbf{mixin.jar},\ \ldots\ \} \qquad (7)\end{aligned}$$

The key to this bootstrapping step is the ability to compose JAR files. By adding a **jarcomposer** to the bootstrapping JARs, we can use the unrefined **lib** to evaluate **(7)**, and in principle can now add any number of new artifact composition tools to AHEAD.

Our **jarcomposer** is simple: it unjars the base JAR file and refinement JARs into distinct directories. It then uses the **composer** bootstrapping JAR to compose these directories, forming their union. The contents of the resulting directory are then placed into a new JAR file. In **(7)**, this JAR file (i.e. $\mathbf{lib_{new}}$**/composer.jar**) becomes the refined **composer** bootstrapping JAR. At this point, $\mathbf{lib_{new}}$ can compose an enlarged set of artifact types (e.g., **XAK** artifacts), thus allowing **ATS** features to be composed.

3.3 Step 4: Producing APL-Specific Products

Once $\texttt{ATS}_{\texttt{bin}'}$ and **APL** were developed, it was a simple matter to define, synthesize, and build different versions of **ATS**. Over one hundred different versions are possible (see Figure 8). For example, a version **K** that contained the **core** and **guidsl** tools is specified and built by:

$$\texttt{ATS}_{\texttt{Ksrc}} = \texttt{guidsl} \bullet \texttt{core} \quad (8)$$

$$\texttt{ATS}_{\texttt{Kbin}} = \texttt{antBuild(}\ \texttt{ATS}_{\texttt{Ksrc}}\ \texttt{)} \quad (9)$$

In the synthesis of $\texttt{ATS}_{\texttt{Ksrc}}$, makefiles and HTML documents of **core** were refined, and the code and document files for **guidsl** were added to **core**. Figure 11 displays part of the synthesized **index.html** document of $\texttt{ATS}_{\texttt{K}}$, which shows a document index for all the tools of **core** (**composer**, **jampack**, **mixin**, **unmixin**, and **jak2java**) and the **guidsl** tool (red dotted box). That is, only documents for tools that are present in $\texttt{ATS}_{\texttt{Ksrc}}$ are listed; no other tool documents are included. This customization is the result of refactoring **ATS**. The evaluation of **(8)** and **(9)** is accomplished using the tools in $\texttt{ATS}_{\texttt{bin}'}$.

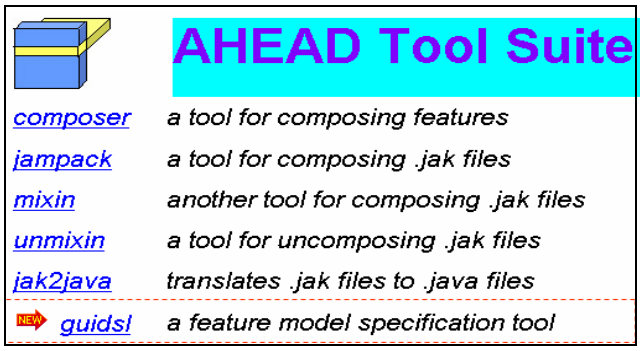


Figure 11. Customized HTML Documentation

4 Lessons Learned and Future Tool Support

We encountered many problems during the refactoring of **ATS**. Most were not related with **ATS**, but with the decomposition process itself, and these problems apply to the refactoring of programs in general. Solutions to these problems and tool support to simplify future refactoring tasks are discussed below.

4.1 Lessons Learned

The greatest challenge in feature refactoring is understanding the original program. When a program grows beyond a few thousand lines of code, it becomes hard to understand what artifacts are impacted by individual features and what dependencies exist among features. Refactoring adds architectural knowledge to a design that was never documented or that was lost. Minimal knowledge of the program and its structure is essential, and we had such knowledge prior to refactoring **ATS**, i.e., we knew approximately the **ATS** directory structure and organization of makefiles and documents.

Dependencies Among Features. Dependencies among features are not always evident. For example, the **bctools** feature requires the **mmatrix** feature; we discovered this requirement when **bctools** failed to compile without **mmatrix**. Once we recognized this relationship, we remembered it by adding a cross-tree constraint to our feature model (Figure 9). Discovering such dependencies and updating feature models is a manual process that we feel could largely be automated.

Error Exposure. Feature refactoring may expose existing errors. For instance, the documentation for the **cpp** tools was not referenced by the **ATS** documentation home page and consequently was not accessible. (A link to the documentation of all tools should appear in the home page). We detected this error during the creation of the **cpp** feature, as we knew the home page document had to be refined, but was not.

Program Extensions. AHEAD and GenVoca are generalizations of object-oriented frameworks [4]. Extending a framework involves adding new classes and extending hook methods that are defined in abstract classes of a framework. While frameworks have been developed primarily for code, AHEAD generalizes this idea to frameworks to non-code artifacts as well. That is, there are variation points in documents of all types, and document extensions are made at these points. It is well-known that manually extending frameworks is error prone, and could be substantially helped by tools that guide users in how to properly extend frameworks [18]. The same ideas apply to non-code documents as well.

The lack of documentation or tool support for adding tools to **ATS** hampered our abilities to feature refactor **ATS**. Extending **ATS** has historically been ad hoc, where variations arise due to different programmers following different procedures. We now know that a new tool would (a) add source files, (b) add one or more document pages that are linked with existing documentation at predefined points, (c) add new build documents that are linked to existing build script at predefined locations, and (d) regression tests must appear in a designated directory. A standard procedure for extending **ATS** would have significantly helped us in refactoring because we would have known exactly where changes would be made (as opposed to discovering these places later when **ant** builds fail or when synthesized documentation is found to be erroneous).

Accidental Complexities (Grandma's Teeth). Accidental complexity lies at the heart of many problems in software engineering [9]. It is unnecessary and arbitrary complexity that obscures the similarity of designs, patterns, and processes that could otherwise be unified. Years ago, we were building different tools for the same language (e.g., pretty-printers, composers, translators), and discovered the process by which each tool was designed and created was unique. Odd details whose only rationale was "that was the way we did it" distinguished different tools. Called "Grandma's teeth", meaning a gross lack of alignment in an otherwise identical design, was an indicator of accidental complexity. By standardizing designs (which reduces complexity), we were able to significantly simplify the design and synthesis of tool suites [5], and make the process of design and implementation more rigorous, structured, repeatable, and streamlined.

Our refactoring of **ATS** exposed other forms of Grandma's teeth that we were unaware of. Makefiles passed parameters to other makefiles, but the same parameter was given different names in different places. Classpaths were defined in different makefiles in different manners. The place where makefiles are altered to add build scripts for similar tools varied significantly. The removal of

accidental complexity and the alignment of similar concepts would solve this, and would substantially simplify program refactoring.

Generality of Experiences. We believe that there is nothing special about `ATS` or our procedure to refactor it. (Although `ATS` is unusual in that it is bootstrapped, this made it *harder* to refactor. Few applications require bootstrapping). The incremental way in which we refactored features from `ATS` is analogous to aspect mining and refactoring (see Section 5), and is not unusual. Further, although the dependencies among features were minimal, the basic approach in Section 3 would be our starting point for future refactorings.

Even though our work is preliminary, we believe that it is possible to feature refactor an application with minimal knowledge of its structure. That we had regression tests per feature helped substantially in validating our efforts. Tool support (discussed in the next section) would also help greatly in a refactoring process.

4.2 Future Tool Support

A Tool for Initial Refactoring. We learned from `ATS` that when a feature encapsulates one or more tools, most artifacts of a feature are new files. Only specific artifacts (makefile gateways, documentation home pages, etc.) are refined. A simple tool could be created to partition the files of a composite feature (i.e., directory) into a pair of directories (representing the contents of different features), thus simplifying an important step in feature refactoring.

Artifact-Specific Refactoring Tools. `ATS` currently has tools for refactoring Java/Jak source classes [22]. Exactly the same kind of tool would be needed for refactoring XML (e.g., `XAK`) artifacts into base and refinement files. Other artifact-specific tools would be needed for refactoring other documents (e.g., Word files, JPEG images).

Safe Composition Tools. A recently added set of tools to the `ATS` arsenal are those that support safe composition [8]. *Safe composition* is the guarantee that programs composed from features are absent of references to undefined elements (such as classes, methods, and variables). Existing safe composition tools are targeted to Java and Jak source files. However, the same concept holds for other artifact types. XML and HTML define elements that can be subsequently referenced. We want to synthesize documents of all types that are devoid of references to undefined elements, including cross-references to elements of documents in different types. Safe composition tools could also be useful in detecting unreferenced elements or benign references. A *benign reference* is a reference to a non-existent file, but no error is reported. `ant` makefiles allow benign references in `fileset` definitions: if a listed file in a `fileset` is not present, `ant` does not complain. Benign references and unreferenced elements suggest (a) they are no longer needed and should be removed, (b) they are not linked to other elements (the topic of Error Exposure of the previous section), or (c) they belong to other features.

In general, we believe that a tremendous amount of automated support for feature refactoring could be provided by detecting unused element definitions or benign references.

5 Related Work

Three SPL adoption models have been proposed, namely, extractive, reactive, and proactive [12]. ATS refactoring exemplified the extractive approach which "*reuses one or more existing software products for the product line's initial baseline*". Few experiences have been published on the extractive approach for SPLs. In [11], re-engineering techniques are used to obtain an SPL from already available programs. The Unix `diff` command is used to extract differences among assets. Pairs of files are compared to obtain the lines matched, lines inserted, lines deleted and lines replaced. These hints are then used to ascertain common abstractions, and to group assets with a shared common structure, code and functionality. Similar concerns are addressed in [10] where the focus is on abstraction elicitation in SPLs; the approach looks for cut-copy-paste clones within distinct pieces of code that can be moved into an abstract superclass. `Diff`-like facilities are also used for this purpose.

These efforts aim at improving reuse, modularity and legibility of the software artifacts. By contrast, feature refactoring is not only concerned about reuse but on engineering a system for variability. Moreover, and unlike prior research, our work underlines the importance and necessity of refactoring and encapsulating multiple representations of programs in features. Our work can be seen as an instance of a general approach to feature-oriented refactoring of legacy programs [22].

Features can use aspects to implement program refinements. Feature refactoring is thus related to aspect mining and refactoring [25][28], which strives to surface hidden concerns, much like our goal to strive to surface particular features [19]. We see three distinctions between our work and aspect refactoring. First is the scale on which we are refactoring: `ATS` features encapsulate tools or groups of tools; *very* few pieces of simple advice (i.e., refinements of designated variation points) are used. Second, we are largely refactoring artifacts other than Java code. And third, while aspect-based approaches rely on labeling methods and inferring other labels via program analysis [26], the approach described in this paper relies on regression tests and build scripts to surface features. Through failures in a build process, the missing pieces are identified and moved to the appropriate feature. This is more akin with SPLs where the production process (basically supported through *build* scripts) plays a major role.

Feature refactoring includes three main activities, namely, feature identification, feature refactoring as such, and refactoring validation. In our approach, the first two are mainly manual whereas regression tests are used to validate the result and, if a build fails, to guide refactoring. A more intensive use of regression tests are presented in [24]. First, test cases are grouped to identify a feature implementation. Clustering and textual pattern analysis is conducted to find test-case clusters. A cluster execution serves to locate source code that implements the feature through the use of code profilers. Once a feature implementation is located, the code is refactored (e.g. global variables are removed and implicit communication is moved to explicit interfaces) into fine-grained components.

6 Conclusions

Feature refactoring is the process of decomposing a legacy program into a set of building blocks called features. Each feature implements an increment in program functionality; it encapsulates new artifacts (code, documentation, regression tests, etc.) that it adds to a program, in addition to the changes it makes to existing artifacts that integrate the new artifacts into a coherent whole. The result of feature refactoring is a software product line, where different variations of the original program can be synthesized by composing different features.

The AHEAD Tool Suite (**ATS**) is the largest program, by almost two orders of magnitude, that we have feature refactored. **ATS** consists of 24 different tools expressed in over 200K LOC Java. We feature refactored **ATS** so that hundreds of **ATS** variants (with or without specific tools) could be synthesized. We expressed the process of refactoring by a simple mathematical model that relates algebraic factoring to artifact refactoring, and program synthesis to expression evaluation. Refining and composing XML documents was critical to our work, and we were able to verify a correct feature refactoring by successfully executing **ATS** build scripts and running regression tests for all synthesized **ATS** tools. We showed how an integral part of refactoring **ATS**, namely its need for bootstrapping, could be explained by refinements. And most importantly, our work revealed generic problems, solutions, and an entire suite of tools that could be created to simplify future feature refactoring tasks.

Our work is a valuable case study on the scalability of feature-based program refactoring and synthesis. We believe our work outlines a new generation of useful program refactoring tools that can simplify future feature refactoring efforts.

Acknowledgments

We gratefully acknowledge the comments of F.I. Anfurrutia, S. Apel, D. Benavides, R. Capilla, C. Lengauer, and R. Lopez-Herrejon. Thanks also to the reviewers. This work was co-supported by the Spanish Ministry of Science and Education under contract #TIC2005-05610 and by NSF's Science of Design Project #CCF-0438786. Trujillo has a doctoral grant from the Spanish Ministry of Science & Education. This work was done while Trujillo was visiting the University of Texas at Austin.

References

[1] AHEAD Tool Suite. **http://www.cs.utexas.edu/users/schwartz/ATS.html**

[2] Apache Struts. **http://struts.apache.org/**

[3] D. Batory, G. Chen, E. Robertson, and T. Wang. "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE TSE*, May 2000.

[4] D. Batory, R. Cardone, and Y. Smaragdakis. "Object-Oriented Frameworks and Product-Lines", *SPLC 2000*.

[5] D. Batory, J. Liu, and J.N. Sarvela. "Refinements and Multi-Dimensional Separation of Concerns", *ACM SIGSOFT 2003*.

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

[7] D. Batory. "Feature Models, Grammars, and Propositional Formulas", *SPLC*, September 2005.

[8] D. Batory and S. Thaker. "Towards Safe Composition of Product Lines", University of Texas Dept. of Computer Sciences TR-06-33. **http://www.cs.utexas.edu/ftp/pub/techreports/index/html/Abstracts.2006.html#TR-06-33**

[9] F.P. Brookes. "No Silver bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, April 1987.

[10] R. Buhrdorf, D. Churchett, C.W. Krueger. "Salion's Experience with a Reactive Software Product Line Approach", *5th Int. Workshop on Software Product-Family Engineering*, LNCS-3014, 2003.

[11] R. Capilla and J.C. Dueñas. "Light-weight Product Lines for Evolution and Maintenance of Web Sites", *7th European Conference On Software Maintenance And Reengineering (CSMR'03)*, 2003.

[12] P. Clements and C. Krueger. Two-part Point/Counterpoint column: "Being Proactive Pays Off" and "Eliminating the Adoption Barrier", *IEEE Software*, July/August 2002.

[13] Cygwin. **http://www.cygwin.com/**

[14] K. Czarnecki, S. Helsen, and U. Eisenecker. "Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models". *Software Process Improvement and Practice*, 10(2), 2005.

[15] A. van Deursen, M. Marin, and L. Moonen. Aspect Mining and Refactoring. *First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*. University of Waterloo, 2003.

[16] O. Diaz, S. Trujillo, and F. I. Anfurrutia. "Supporting Production Strategies as Refinements of the Production Process". *SPLC*, 2005.

[17] O. Diaz, S. Trujillo, and F. I. Anfurrutia. "XAK and XML Refinement." Draft in preparation, April 2006.

[18] G. Froehlich, H.J. Hoover, L. Liu and P.G. Sorenson. "Hooking into Object-Oriented Application Frameworks", *ISCE 1997*.

[19] J. Hannemann and G. Kiczales. "Overcoming the Prevalent Decomposition of Legacy Code", *Workshop on Advanced Separation of Concerns*, *IEEE*, 2001.

[20] J. Liu and D. Batory. "Automatic Remodularization and Optimized Synthesis of Product-Families", *GPCE 2004*.

[21] J. Liu, D. Batory, and S. Nedunuri, "Modeling Interactions in Feature Oriented Designs", *International Conference on Feature Interactions (ICFI)*, June 2005.

[22] J. Liu, D. Batory, and C. Lengauer. "Feature Oriented Refactoring of Legacy Applications", *ICSE*, 2006.

[23] R. Lopez-Herrejon and D. Batory. "From Crosscutting Concerns to Product Lines: A Functional Composition Approach", submitted for publication 2006.

[24] A. Mehta and G.T. Heineman. "Evolving Legacy System Features into Fine-Grained Components", *ICSE*, 2002.

[25] G. C. Murphy, et al. “Separating Features in Source Code: An Exploratory Study”, *ICSE,* 2001.

[26] P. Robillard and G. C. Murphy. “Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies”, *ICSE*, 2002.

[27] 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.

[28] C. Zhang and H.-A. Jacobsen. “Resolving Feature Convolution in Middleware Systems, *OOPSLA*, 2004.