Towards Safe Composition of Product Lines

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

ABSTRACT
Programs of a software product line can be synthesized by composing modules that implement features. Besides high-level domain constraints that govern the compatibility of features, there are also low-level implementation constraints: a feature module can reference elements that are defined in other feature modules. *Safe composition* is the guarantee that programs composed from feature modules are absent of references to undefined elements (such as classes, methods, and variables). We show how many properties of safe composition can be verified for AHEAD product lines using feature models and SAT solvers.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]: Modules and interfaces; D.2.4 [**Software Program Verification**]: Assertion checkers; D.2.11 [**Software Architectures**]: Data abstraction, Languages.

General Terms
Design, Languages, Verification.

Keywords
compositional programming, program synthesis, SAT solvers, features.

1 INTRODUCTION
The essence of software product lines is the systematic and efficient creation of products [13]. Features are commonly used to specify and distinguish members of a product line, where a *feature* is an increment in program functionality. Features communicate product functions in an easy-to-understand way, they capture functionalities concisely, and help delineate commonalities and variabilities in a domain [29].

We have argued that if features are primary entities that describe products, then modules that implement features should also be primary entities in software design and program synthesis. This line of reasoning has lead us to compositional and declarative models of programs in software product lines. A program is declaratively specified by the list of features that it supports. Tools directly translate such a specification into a composition of feature modules that synthesize the target program [6][10].

Not all features are compatible. Feature models or feature diagrams are commonly used to define the legal combinations of features in a product line. In addition to domain constraints, there are low-level implementation constraints that must also be satisfied. For example, a feature module can reference a class, variable, or method that is defined in another feature module. *Safe composition* is the guarantee that programs composed from feature modules are absent of references to undefined classes, methods, and variables. More generally, safe composition is a particular problem of *safe generation*: the guarantee that generators synthesize programs with particular properties [47][51][49][25]. There are few results on safe generation of product lines [33][17].

In this paper, we show how many properties of safe composition can be achieved for AHEAD product lines by using feature models and SAT solvers. We identify properties of safe composition, and report our findings on two different product lines to verify that these properties hold for all product line members. Some properties that we analyze do not reveal actual errors, but rather designs that "smell bad" and that could be improved.

2 FORMAL MODELS OF PRODUCT LINES
A *feature model* is a hierarchy of features that is used to distinguish products of a product line [28][16]. Consider an elementary automotive product line that differentiates cars by transmission type (automatic or manual), engine type (electric or gasoline), and the option of cruise control. A *feature diagram* is a common way to depict a feature model. Figure 1 shows the diagram of this product line. A car has a body, engine, transmission, and optionally a cruise control. A transmission is either automatic or manual (choose one), and an engine is electric-powered, gasoline-powered, or both.

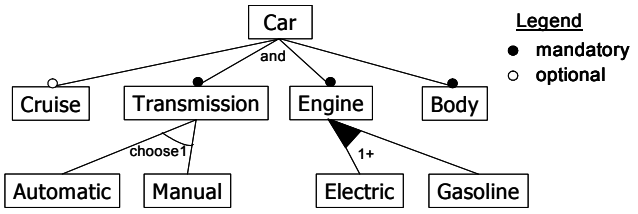


Figure 1 A Feature Diagram

Besides hierarchical relationships, feature models also allow cross-tree constraints. Such constraints are often inclusion or exclusion statements of the form if feature `F` is included in a product, then features `A` and `B` must also be included (or excluded). A cross-tree constraint is that cruise control requires an automatic transmission.

A feature diagram is a graphical depiction of a context-free grammar [27]. Rules for translating feature diagrams to grammars are listed in Figure 2. A bracketed term `[B]` means that feature `B` is optional, and term `S+` means select one or more subfeatures of `S`. We assume subfeature selections are not replicated and the order in which subfeatures appear in a sentence is the order in which they are listed in the grammar [11].

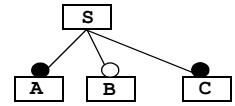
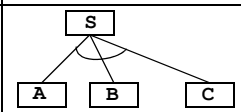
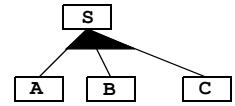
concept	diagram notation	grammar	propositional formula
and		`S : A [B] C ;`	`(S⇔A) ∧ (B⇒S) ∧ (C⇔S)`
alternative (choose1)		`... S ...` `S : A | B | C ;`	`(S ⇔ A ∨ B ∨ C)` `∧ atmost1(A,B,C)`
or (choose 1+)		`... S+ ...` `S : A | B | C ;`	`S ⇔ A ∨ B ∨ C`

Figure 2 Feature Diagrams, Grammars, and Propositional Formulas

A specification of a feature model is a grammar and its cross-tree constraints. A model of our automotive product line is listed in Figure 3. A sentence of this grammar that satisfies all cross-tree constraints defines a unique product and the set of all legal sentences is a language, i.e., a product line [11].

```
// grammar of our automotive product line

Car : [Cruise] Transmission Engine+ Body ;

Transmission : Automatic | Manual ;

Engine : Electric | Gasoline ;

// cross-tree constraints

Cruise ⇒  Automatic ;
```

Figure 3 A Feature Model Specification

We recently showed that feature models are compact representations of propositional formulas [11]. Rules for translating grammar productions into formulas are listed in Figure 2. (The `atmost1(A,B,C)` predicate in Figure 2 means at most one of `A`, `B`, or `C` is true. See [21] p. 278.) The propositional formula of a grammar is the conjunction of the formulas for each production, each cross-tree constraint, and the formula that selects the root feature (i.e., all products have the root feature). Thus, *all constraints except ordering constraints of a feature model can be mapped to a propositional formula*. This relationship of feature models and propositional formulas is essential to results on safe composition.

3 AHEAD

AHEAD is a theory of program synthesis that merges feature models with additional ideas [10]. First, each feature is implemented by a distinct module. Second, program synthesis is compositional: complex programs are built by composing feature modules. Third, program designs are algebraic expressions. The following summarizes the ideas of AHEAD that are relevant to safe composition.

3.1 Algebras and Step-Wise Development

An AHEAD model of a domain is an *algebra* that consists of 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`. One or more features of a model are *constants* that represent base programs:

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

The remaining operations are *functions*, which are program refinements or extensions:

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

where • denotes function composition and `i•x` is read as "feature `i` refines program `x`" or equivalently "feature `i` is added to program `x`". The *design* of an application is a named expression (i.e., composition of features) called an *equation*:

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

AHEAD is based on step-wise development [52]: one begins with a simple program (e.g., constant feature `h`) and builds a more complex program by progressively adding features (e.g., adding features `i` and `j` to `h` in `prog3`).

The relationship between feature models and AHEAD is simple: the operations of an AHEAD algebra are the primitive features of a feature model; compound features (i.e., non-leaf features of a feature diagram) are AHEAD expressions. Each sentence of a feature model defines an AHEAD expression which, when evaluated, synthesizes that product. The AHEAD model `Auto` of the automotive product line is:

```
Auto = { Body, Electric, Gasoline, Automatic,
         Manual, Cruise }
```

where `Body` is the lone constant. Some products (i.e., legal expressions or sentences) of this product line are:

```
c1 = Automatic•Electric•Body
c2 = Cruise•Automatic•Electric•Gasoline•Body
```

`c1` is a car with an electric engine and automatic transmission. And `c2` is a car with both electric and gasoline engines, automatic transmission, and cruise control.

3.2 Feature Implementations

Features are implemented as program refinements. Consider the following example. Let the **BASE** feature encapsulate an elementary buffer class with **set** and **get** methods. Let **RESTORE** denote a "backup" feature that remembers the previous value of a buffer. Figure 4a shows the **buffer** class of **BASE** and Figure 4b shows the **buffer** class of **RESTORE•BASE**. The underlined code indicates the changes **RESTORE** makes to **BASE**. Namely, **RESTORE** adds to the **buffer** class two members, a **back** variable and a **restore** method, and modifies the existing **set** method. While this example is simple, it is typical of features. Adding a feature means adding new members to existing classes and modifying existing methods. As programs and features get larger, features can add new classes and packages to a program as well.

```
class buffer {
  int buf = 0;
  int get() {return buf;}
  void set(int x) {
    buf=x;
  }
}
```
(a)

```
class buffer {
  int buf = 0;
  int get() {return buf;}
  int back = 0;
  void set(int x) {
    back = buf;
    buf=x;
  }
  void restore() {
    buf = back;
  }
}
```
(b)

Figure 4 Buffer Variations

Features can be implemented in many ways. The way it is done in AHEAD is to write program refinements in the Jak language, a superset of Java [10]. The changes **RESTORE** makes to the **buffer** class is a refinement that adds the **back** and **restore** members and refines the **set** method. This is expressed in Jak as:

```
refines class buffer {
  int back = 0;
  void restore() { buf = back; }
  void set(int x) { back = buf; Super.set(x); }
}                                                    (1)
```

Method refinement in AHEAD is accomplished by inheritance; **Super.set(x)** indicates a call to (or substitution of) the prior definition of method **set(x)**. By composing the refinement of **(1)** with the class of Figure 4a, a class that is equivalent to that in Figure 4b is produced. See [10] for further details.

AspectJ could also be used to implement features. As the refinement capabilities of AspectJ are more general than that of method refinement in AHEAD, we delay further discussion of aspect implementations of features until Section 7.

4 "BIG INHALE" COMPILATION

The first step in testing safe composition properties is to provide a global analysis of the feature modules that can be composed. The analysis (a) determines how each class, method, and variable reference in every module binds to a definition, and (b) eliminates or identifies ambiguities and other problems related to module compilation. We used a variation of a technique that was pioneered in Hyper/J for compiling *hyperslices* (i.e., Hyper/J modules) [40]. As an approximation, an AHEAD feature module is a hyperslice. To compile a hyperslice, stubs are created for all classes and members that are not introduced by that hyperslice. This makes them *declaratively complete*. Once stubs are available, the Java classes of a hyperslice can be compiled into bytecode. Hyper/J then uses bytecode composition tools to compose independently compiled hyperslices. We follow a similar approach.

We know of no tool support for automatic stub creation in Hyper/J; stubs must be created manually [36]. An advantage of AHEAD and many product lines is that the source or binaries for all features are available. By analyzing the feature code base (which we call the "big inhale"), we can automatically generate stubs for all classes that could appear in a synthesized product [9]. For every class, we create a stub that contains the union of the signatures of all variables, methods, and declarations that could appear in that class. The same applies to interfaces that a class could implement. Remember a Java class **C** in feature module **M** encapsulates a fragment of a class **P.C** that could appear in a synthesized program **P**. When we compile module **M**, we bind all references in class **C** of **M** to the variables, methods, and classes of our generated stubs. Only at module composition time do we rebind each variable, method, etc. reference in **C** of **M** to a definition, where the definition of a variable, method, or class may be supplied by one of any of the features that comprise **P**.

An important point of feature module compilation is that it provides a global consistency check on modules, without dealing with the feature combinatorics that is the subject of safe composition discussed in the next section. An example of a consistency property is for a feature module **F** to reference a method that is not defined in *any* feature module. We catch this error because module **F** fails to compile.

Ambiguities are another source of errors that our compilation technique catches. Consider the base program **BaseP** in Figure 5a, which consists of two interfaces (**I**,**J**) and three classes (**X**,**A**,**B**). **BaseP** seems consistent in isolation: the **foo(x)** call in Figure 5a binds to the **foo(I)** method of class **A**. Now consider feature module **ExtendX** of Figure 5b that makes class **X** also implement interface **J**. This global knowledge is exposed by our class stubs, and module **BaseP** fails to compile as a consequence: the **foo(x)** call is ambiguous as it could be bound to either the **foo(I)** or **foo(J)** methods of class **A**.

```
interface I {}
interface J {}
class X implements I {}
class A {
  void foo(I b) {}
  void foo(J d) {}
}
class B {
  void bar(A a,X x) {
     a.foo(x);
  }
}
```
(a)

```
refines class X
    implements J {}
```
(b)

Figure 5 Uncompilable Feature Modules

In the following sections, assume we have the bytecodes of each feature module from which we can extract variable, method, and class and interface references.

5 SAFE COMPOSITION

The AHEAD tool suite has multiple ways to compose feature modules to build a product. We can compile feature modules as discussed in the last section and let AHEAD compose their bytecodes to produce the binary of a product directly. A problem that can arise is that there may be references to classes or members that are undefined. Alternatively, the primary way in which features are composed in AHEAD is by composing source files. But now, the same errors (i.e., reference to undefined elements) are discovered at program compilation time. In short, we need to ensure that all variables, methods, and classes that are referenced in a program are indeed defined. And we want to ensure this property for all programs in a product line, regardless of the specific approach to synthesize products. This is the essence of safe composition.

The core problem is illustrated in the following example. Let **PL** be a product line with three features: **base**, **addD**, and **refC**. Figure 6 shows their modules. **base** is a base feature that encapsulates class **C** with method **foo()**. Feature **addD** introduces class **D** and leaves class **C** unchanged. Feature **refC** refines method **foo()** of class **C** and references the constructor of class **D**. Now suppose the feature model of **PL** is a single production with no cross-tree constraints:

```
class C {
  void foo(){..}
}
```
(a) **base**

```
class D {...}
```
(b) **addD**

```
refines class C {
  void foo(){
    ... new D() ...
    Super.foo();
  }
}
```
(c) **refC**

Figure 6 Three Feature Modules

```
PL : [refC] [addD] base ; // feature model
```

The product line of **PL** has four programs that represent all possible combinations of the presence/absence of the **refC** and **addD** features. All programs in **PL** use the **base** feature. Question: are there programs in **PL** that have type errors? As **PL** is so simple, it is not difficult to see that there is such a program: it has the AHEAD expression **refC•base**. Class **D** is referenced in **refC**, but there is no definition of **D** in the program itself. This means one of several possibilities: the feature model is wrong, feature implementations are wrong, or both. Designers need to be alerted to such errors. In the following, we define some general compositional constraints (i.e., properties) that product lines must satisfy.

5.1 Properties of Safe Composition

Refinement Constraint. Suppose a member or class **m** is introduced in features **X**, **Y**, and **Z**, and is refined by feature **F**. Products in a product line that contain feature **F** must satisfy the following constraints to be type safe:

> (i) **X**, **Y**, and **Z** must appear prior to **F** in the product's AHEAD expression (i.e., **m** must be defined prior to be refined), and
> (ii) at least **X**, **Y**, or **Z** must appear in every product that contains feature **F**.

Property (i) can be verified by examining the feature model, as it linearizes features. Property (ii) requires the feature model (or rather its propositional formula) to satisfy the constraint:

$$F \Rightarrow X \vee Y \vee Z \qquad (2)$$

By examining the code base of feature modules, it is possible to identify and collect such constraints. These constraints, called *implementation constraints*, are a consequence of feature implementations, and may not arise if different implementations are used. Implementation constraints can be added to the existing cross-tree constraints of a feature model and obeying these additional constraints will guarantee safe composition. That is, only programs that satisfy domain *and* implementation constraints will be synthesized. Of course, the number of implementation constraints may be huge for large programs. However, a majority of constraints will be redundant. Theorem provers, such as Otter [5], could be used to prove that implementation constraints are implied by the feature model and thus can be discarded.

Czarnecki [17] recently observed the following: Let PL_f be the propositional formula of product line **PL**. If there is a constraint **R** that is to be satisfied by all members of **PL**, then the formula ($PL_f \wedge \neg R$) can not be satisfiable. If it is, we know that there is a product of **PL** that violates **R**. To make our example concrete, to verify that a product line **PL** satisfies property **(2)**, we want to prove that all products of **PL** that use feature **F** also use **X**, **Y**, or **Z**. A *satisfiability (SAT)* solver can verify if ($PL_f \wedge F \wedge \neg X \wedge \neg Y \wedge \neg Z$) is satisfiable. If it is, there exists a product that uses **F** without **X**, **Y**, or **Z**. The variable bindings that are returned by a solver identifies the offending product. In this manner, we can verify that all products of **PL** satisfy **(2)**.

> Note: We are inferring composition constraints for each feature module; these constraints lie at the module's "requires-and-provides interface" [18]. When we compose feature modules, we must verify that their "interface" constraints are satisfied by a composition. If composition is a linking process, we are guaranteeing that there will be no linking errors.

Superclass Constraint. **Super** has multiple meanings in the Jak language. The original intent was that **Super** would refer to the method that was being refined. Once a method **void m()** in a class **C** is defined, it is refined by a specification of the form:

```
void m() {... Super.m(); ... }          (3)
```

(In AOP-speak, **(3)** is an around method for an execution pointcut containing the single joinpoint of the **m()** method). However, if no method **m()** exists in class **C**, then **(3)** is interpreted as a method introduction that invokes its corresponding superclass method. That is, method **m()** is added to **C** and **Super.m()** invokes **C**'s inherited method **m()**. To test the existence of a superclass method requires a more complex constraint.

Let feature **F** introduce a method **m** into class **C** and let **m** invoke **m()** of its superclass. Let H_n be a superclass of **C**, where **n** indicates the position of H_n by the number of ancestors above **C**. Thus H_0 is class **C**, H_1 is the superclass of **C**, H_2 is the super superclass of **C**, etc. Let Sup_n**(m)** denote the predicate that is the disjunction of all

features that define method m in H_n (i.e., m is defined with a method body and is not abstract). If features X and Y define m in H_1, then $Sup_1(m)=X \vee Y$. If features Q and R define m in H_2, then $Sup_2(m)=Q \vee R$. And so on. The constraint that m is defined in some superclass is:

$$F \Rightarrow Sup_1(m) \vee Sup_2(m) \vee Sup_3(m) \vee \ldots \quad (4)$$

In short, if feature F is in a product, then there must also be some feature that defines m in a superclass of C. The actual predicate that is used depends on C's position in the inheritance hierarchy.

> Note: it is common for a method n() of a class C to invoke a different method m() of its superclass via Super.m(). Constraint (4) is also used to verify that m() is defined in a superclass of C.

Reference Constraint. Let feature F reference member m of class C. This means that some feature must introduce m in C or m is introduced in some superclass of C. The constraint to verify is:

$$F \Rightarrow Sup_0(m) \vee Sup_1(m) \vee Sup_2(m) \vee \ldots \quad (5)$$

> Note: By treating Super calls as references, (5) subsumes constraints (2) and (4).

> Note: a special case of (5) is the following. Suppose C is a direct subclass of class S. If C is introduced in a product then S must also be introduced. Let c be the default constructor of C which invokes the default constructor m of S. If feature F introduces C and features X, Y, and Z introduce S, then (5) simplifies to:

$$F \Rightarrow Sup_0(m) \quad \text{// same as } F \Rightarrow X \vee Y \vee Z \quad (6)$$

Single Introduction Constraint. More complicated properties can be verified in the same manner. An example is when the same member or class is introduced multiple times in a composition, which we call *replacing*. While not necessarily an error, replacing a member or class can invalidate the feature that first introduced this class or member. For example, suppose feature A introduces the Value class, which contains an integer member and a get() method (Figure 7a). Feature B replaces — not refines — the get() method by returning the double of the integer member (Figure 7b). Both A and B introduce method get(). Their composition, B•A, causes A's get method to be replaced by B's get (see Figure 7c). If subsequent features depend on the get() method of A, the resulting program may not work correctly.

```
class Value {
   int v;
   int get()
   { return v; }
}
```
(a) A

```
refines class Value {
   int get()
   { return 2*v; }
}
```
(b) B

```
class Value {
   int v;
   int get()
   { return 2*v; }
}
```
(c) B•A

Figure 7 Overriding Member

It is possible for multiple introductions to be correct; in fact, we carefully used such designs in building AHEAD. More often, such designs are symptomatic of inadvertent captures [31]: a member is inadvertently named in one feature identically to that of a member in another feature, and both members have different meanings. In general, these are "bad" designs that could be avoided with a more structured design where each member or class is introduced precisely once in a product. Testing for multiple introductions can either alert designers to actual errors or to designs that "smell bad". We note that this problem was first recognized by Flatt et al in mixin compositions [19], and has resurfaced elsewhere in object delegation [30] and aspect implementations [4].

Suppose member or class m is introduced by features X, Y, and Z. The constraint that no product has multiple introductions of m is:

```
atmost1(X,Y,Z) // at most one of X,Y,Z is true (7)
```

The actual constraint used depends on the features that introduce m.

Abstract Class Constraint. An abstract class can define abstract methods (i.e., methods without a body). Each concrete subclass C that is a descendant of an abstract class A must implement all of A's abstract methods. To make this constraint precise, let feature F declare an abstract method m in abstract class A. (F could refine A by introducing m, or F could introduce A with m). Let feature X introduce concrete class C, a descendant of A. If F and X are compatible (i.e., they can appear together in the same product) then C must implement m or inherit an implementation of m. Let C.m denote method m of class C. The constraint is:

$$F \wedge X \Rightarrow Sup_0(C.m) \vee Sup_1(C.m) \vee Sup_2(C.m) \vee \ldots \quad (8)$$

That is, if abstract method m is declared in abstract class A and C is a concrete class descendant of A, then some feature must implement m in C or an ancestor of C.

> Note: to minimize the number of constraints to verify, we only need to verify (8) on concrete classes whose immediate superclass is abstract; A need not be C's immediate superclass.

> Note: Although this does not arise in the product lines we examine later, it is possible for a method m that is abstract in class A to override a concrete method m in a superclass of A. (8) would have to be modified to take this possibility into account.

Interface Constraint. Let feature F refine interface I by introducing method m or that F introduces I which contains m. Let feature X either introduce class C that implements I or that refines class C to implement I (i.e., a refinement that adds I to C's list of implemented interfaces). If features F and X are compatible, then C must implement or inherit m. Let C.m denote method m of class C. The constraint is:

$$F \wedge X \Rightarrow Sup_0(C.m) \vee Sup_1(C.m) \vee Sup_2(C.m) \vee \ldots \quad (9)$$

This constraint is identical in form to (8), although the parameters F, X, and m may assume different values.

5.2 Perspective

We identified six properties ((2),(4)-(9)) that are essential to safe composition. We believe these are the primary properties to

check. We know that there are other constraints that are particular to AHEAD that could be included; some are discussed in Section 7.1. Further, using a different compilation technology may introduce even more constraints to be checked (see Section 7.2).

To determine if we have a full compliment of constraints requires a theoretical result on the soundness of the type system of the Jak language, which is a superset of Java. To place such a result into perspective, we are not aware of a proof of the soundness of the entire Java language. A standard approach for soundness proofs is to study a representative subset of Java, such as Featherweight Java [26] or ClassicJava [20]. Given a soundness proof, it should be possible to determine if any constraints are missing for that language subset. To do this for Jak is a topic of future work.

5.3 Beyond Code Artifacts

The ideas of safe composition transcend code artifacts [17]. Consider an XML document; it may reference other XML documents in addition to referencing internal elements. If an XML document is synthesized by composing feature modules [10], we need to know if there are references to undefined elements or files in these documents. Exactly the same techniques that we outlined in earlier sections could be used to verify safe composition properties of a product line of XML documents. We believe the same holds for product lines of other artifacts (grammars, makefiles, etc.) as well. The reason is that we are performing analyses on *structures* that are common to all kinds of synthesized documents; herein lies the generality and power of our approach.

6 RESULTS

We have analyzed the safe composition properties of many different AHEAD product lines. Table 1 summarizes the key size statistics for several of the product lines that we analyzed. For lack of space in this paper, we report the specifics of the first two product lines listed (PPL and BPL). Note that the size of the code base and and average size of a generated program is listed both in Jak LOC and translated Java LOC.

Product Line	# of Features	# of Programs	Code Base Jak/Java LOC	Program Jak/Java LOC
PPL	7	20	2000/2000	1K/1K
BPL	17	8	12K/16K	8K/12K
GPL	18	80	1800/1800	700/700
JPL	70	56	34K/48K	22K/35K

Table 1: Product Line Statistics

The properties that we verified are grouped into five categories:

- **Refinement** `(2)`,
- **Reference to Member or Class** includes `(4)` and `(5)`
- **Single Introduction** `(7)`
- **Abstract Class** `(8)`
- **Interface** `(9)`.

For each constraint, we generate a theorem to verify that all products in a product line satisfy that constraint. We report the number of theorems generated in each category. Note that duplicate theorems can be generated. Consider features `Y` and `ExtendY` of

```
class D {
  static int i;
  static void o() {..}
  void p() {..}
}
```

(a) `Y`

```
class C {
  void m() { D.o(); }
}

refines class D {
  void p() {
    Super.p();
    D.i=2;
  }
}
```

(b) `ExtendY`

Figure 8 Sources of `ExtendY`⇒`Y`

Figure 8. Method `m` in `ExtendY` references method `o` in `Y`, method `p` in `ExtendY` references field `i` in `Y`, and method `p` in `ExtendsY` refines method `p` defined in `Y`. We create a theorem for each constraint; all theorems are of the form `ExtendY`⇒`Y`. We eliminate duplicate theorems, and report only the number of failures per category. If a theorem fails, we report all (in Figure 8, all three) sources of errors. Finally, we note that very few abstract methods and interfaces were used in the product lines of Table 1. So the numbers reported in the last two categories are small.

We conducted our experiments on a Mobile Intel Pentium 2.8 GHz PC with 1GB memory running Windows XP. We used J2SDK version 1.5.0_04 and the SAT4J Solver version 1.0.258RC [45].

6.1. Prevayler Product Line

Prevayler is an open source application written in Java that maintains an in-memory database and supports plain Java object persistence, transactions, logging, snapshots, and queries [43]. We refactored Prevayler into the *Prevaler Product Line (PPL)* by giving it a feature-oriented design. That is, we refactored Prevayler into a set of feature modules, some of which could be removed to produce different versions of Prevayler with a subset of its original capabilities. Note that the analyses and errors we report in this section are associated with our refactoring of Prevayler into PPL, and not the original Prevayler source[1].

The code base of the PPL is 2029 Jak LOC with seven features:

- **Core** — This is the base program of the Prevayler framework.
- **Clock** — Provides timestamps for transactions.
- **Persistent** — Logs transactions.
- **Snapshot** — Writes and reads database snapshots.
- **Censor** — Rejects transactions by certain criteria.
- **Replication** — Supports database duplication.
- **Thread** — Provides multiple threads to perform transactions.

A feature model for Prevayler is shown in Figure 9. Note that there are constraints that preclude all possible combinations of features.

```
// grammar
PREVAYLER : [Thread] [Replication] [Censor]
            [Snapshot] [Persistent] [Clock] Core ;

//constraints
Censor ⇒ Snapshot;
Replication ⇒ Snapshot;
```

Figure 9. Prevayler Feature Model

1. We presented a different feature refactoring of Prevayler in [37]. The refactoring we report here is similar to an aspect refactoring of Godil and Jacobsen [22].

Results. The statistics of our PPL analysis is shown in Table 2. We generated a total of 882 theorems, of which 791 were duplicates. To analyze the PPL feature module bytecodes, generate and remove duplicate theorems, and run the SAT solver to prove the 91 unique theorems took 8 seconds.

We performed two sets of safe composition tests on Prevalyer. In the first test, we found 15 reference constraint violations, of which 8 were unique errors, and 12 multiple-introduction constraint errors. These failures revealed an omission in our feature model: we were missing a constraint "**`Replication` ⇒ `Snapshot`**". After changing the model (to that shown in Figure 9) we found 11 reference failures, of which 4 were unique errors, and still had 12 multiple-introduction failures. These are the results in Table 2.

Constraint	# of Theorems	Failures
Refinement	39	0
Reference to Member or a Class	830	11
Single Introduction	12	12
Abstract Class	0	0
Interface	1	0

Table 2: Prevayler Statistics

Two reference failures were due to yet another error in the feature model that went undetected. Feature **`Clock`** must not be optional because all other features depend on its functionality. We fixed this by removing **`Clock`**'s optionality.

A third failure was an implementation error. It revealed that a code fragment had been misplaced — it was placed in the **`Snapshot`** where it should have been placed in **`Replication`**. The last failure was similar. A field member that only **`Thread`** feature relied upon, was defined in the **`Persistent`** feature, essentially making **`Persistent`** non-optional if **`Thread`** is selected. The error was corrected by moving the field member into **`Thread`** feature.

Making the above-mentioned changes resolved all reference constraint failures, but 12 multiple-introduction failures remained. They were not errors, rather "bad-smell" warnings. Here is a typical example. **`Core`** has the method:

```
public TransactionPublisher publisher(..) {
  return new CentralPublisher(null, ...);
}
```

`Clock` replaces this method with:

```
public TransactionPublisher publisher(..) {
  return new CentralPublisher(new Clock(), ...);
}
```

Alternatively, the same effect could be achieved by altering the **`Core`** to:

```
ClockInterface c = null;
public TransactionPublisher publisher(..) {
  return new CentralPublisher(c, ...);
}
```

And changing **`Clock`** to refine **`publisher()`**:

```
public TransactionPublisher publisher(..) {
  c = new Clock();
  return Super.publisher(..);
}
```

Our safe composition checks allowed us to confirm by inspection that the replacements were performed with genuine intent.

6.2 Bali

The *Bali Product Line (BPL)* is a set of AHEAD tools that manipulate, transform, and compose AHEAD grammar specifications [10]. The feature model of Bali is shown in Figure 10. It consists of 17 primitive features and a code base of 8K Jak (12K Java) LOC plus a grammar file from which a parser can be generated. Although the number of programs in BPL is rather small (8), each program is about 8K Jak LOC or 12K Java LOC that includes a generated parser. The complexity of the feature model of Figure 10 is due to the fact that our feature modelling tools preclude the replication of features in a grammar specification, and several (but not all) Bali tools use the same set of features.

```
Bali    : Tool [codegen] Base ;

Base    : [require] [requireSyntax] collect
                visitor bali syntax kernel;

Tool    : [requireBali2jak] bali2jak
        | [requireBali2jcc] bali2jcc
        | [requireComposer] composer
        | bali2layerGUI bali2layer
                bali2layerOptions ;
%%
composer ⇒  ¬codegen;
bali2jak ∨ bali2layer ∨ bali2javacc ⇔ codegen;
bali2jak ∧ require ⇒  requireBali2jak; // 1
bali2jcc ∧ require ⇒  requireBali2jcc; // 2
composer ∧ require ⇒  requireComposer; // 3
require ⇒  requireSyntax;
```

Figure 10 Bali Feature Model

The statistics of our BPL analysis is shown in Table 3. We generated a total of 3453 theorems, of which 3358 were duplicates. To analyze the BPL feature module bytecodes, generate and remove duplicate theorems, and run the SAT solver to prove the 95 unique theorems took 4 seconds.

Constraint	# of Theorems	Failures
Refinement	42	0
Reference to Member or a Class	3334	7
Single Introduction	18	7
Abstract Class	41	0
Interface	18	0

Table 3: Bali Product Line Statistics

We found several failures, some of which were due to duplicate theorems failing, and the underlying cause boils down to two errors. The first was a unrecognized dependency between the **`requireBali2jcc`** feature and the **`require`** feature, namely

requireBali2jcc invokes a method in **require**. The feature model of Figure 10 allows a Bali tool to have **requireBali2javacc** without **require**. A similar error was the **requireComposer** feature invoked a method of the **require** feature, even though **require** need not be present. These failures revealed an error in our feature model. The fix is to replace rules 1-3 in Figure 10 with:

```
Bali2JakTool ⇒ (require ⇔ requireBali2jak); // new 1
Bali2jccTool ⇒ (require ⇔ requireBali2jcc); //new 2
BaliComposerTool ⇒ (require ⇔ requireComposer);// new 3
```

We verified that these fixes do indeed remove the errors.

Another source of errors deals with replicated methods (i.e., multiple introductions). When a new feature module is developed, it is common to take an existing module as a template and rewrite it as a new module. In doing so, some methods are copied verbatim and because we had no analysis to check for replication, replicas remained. Since the same method overrides a copy of itself in a composition, no real error resulted. This error revealed a "bad smell" in our design that has a simple fix — remove replicas.

We found other multiple introductions. The **kernel** feature defines a standard command-line front-end for all Bali tools. To customize the front-end to report the command-line options of a particular tool, a **usage()** method is refined by tool-specific features. In some tools, it was easier to simply override **usage()**, rather than refine it with a tool-specific definition. In another case, the overriding method could easily have been restructured to be a method refinement. In both cases, we interpreted these failures as "bad smell" warnings and not true errors.

6.3 Other Product Lines

We have evaluated other product lines w.r.t. safe composition properties. Some of these product lines were considerably larger than those presented in previous sections. The results were similarly encouraging: product lines whose code base is close to 50K Java LOC and whose programs are 35K Java LOC apiece took under 30 seconds to analyze.

7 RELATED AND FUTURE WORK

7.1 Other Safe Composition Constraints

The importance of ordering features in a composition can be limited to defining a class or method prior to refining it. We verified these requirements in our product lines and found no errors. However, it is possible in AHEAD to write feature modules that *reference* methods that are added by subsequently composed features. Here is a simple example.

The modules for features **Base** and **Ref** are shown in Figure 11. **Base** encapsulates class **C** that has a method **foo()** which invokes method **bar()**. Module **Ref** refines class **C** by introducing method **bar()**. The composition **Ref•Base** is shown in Figure 11c.

Observe that **Ref•Base** (Figure 11c) has no references to undefined members. But the design of **Base** does not strictly follow the requirements of stepwise development, which asserts after each step in a program's development, there should be an absence of references to undefined members. **Base** does not have this property (i.e., it fails to satisfy the Reference constraint **(5)** as **bar()** is undefined).

```
class C {
  void foo() {
    bar();
  }
}
```
(a) **Base**

```
refines class C {
  bar() {..}
}
```
(b) **Ref**

```
class C {
  void foo() {
    bar();
  }
  void bar() {..}
}
```
(c) **Ref•Base**

Figure 11 Ordering Example

We can circumvent this problem by rewriting **Base** as **Base1** in Figure 12, where **Base1** includes an empty **bar()** method. Composing **Base1** with **Ref** overrides the **bar()** method, and the composition of **Ref•Base1** is again class **C** of Figure 11c. Now both expressions **Base1** and **Ref•Base1** satisfy our safe composition properties. Unfortunately this revised design raises the warning of multiple introductions.

```
class C {
  void foo() {
    bar();
  }
  void bar(){};
}
```

Figure 12 Improved Base Design

Satisfying the strict requirements of stepwise development is not essential for safe composition. Nevertheless, it does lead to another set of interesting automated analyses and feature module refactorings that are subjects of future work.

7.2 Related Work

Undefined methods and classes can arise in the linking or run-time loading of programs when required library modules cannot be found [38]. Our work addresses a variant of this problem from the perspective of product lines and program generation.

Safe generation is the goal of synthesizing programs with specific properties. Although the term is new [25], the problem is well-known. The pioneering work of Goguen, Wagner, et al using algebraic specifications to create programs [51], and the work at Kestrel [47] to synthesize programs from formal models are examples. Synthesis and property guarantees of programs in these approaches require sophisticated mathematical machinery. AHEAD relies on simple mathematics whose refinement abstractions are virtually identical to known OO design concepts (e.g., inheritance).

MetaOCaml adds code quote and escape operations to OCaml (to force or delay evaluation) and verifies that generated programs are well-typed [49]. Huang, Zook, and Smaragdakis [25] studied safe generation properties of templates. Templates are written in a syntax close to first-order logic, and properties to be verified are written similarly. Theorem provers verify properties of templates. Our work is different: feature modules are a component technology where we verify properties of component compositions. The clos-

est research to ours, and an inspiration for our work, is that of Czarnecki and Pietroszek [17]. Unlike our work, they do not use feature modules. Instead, they define an artifact (e.g., specification) using preprocessor directives, e.g., an element is included in a specification if a boolean expression is satisfied. The expression references feature selections in a feature model. By defining constraints on the presence or absence of an element, they can verify that a synthesized specification for all products in a product line is well-formed. Our work on safe composition is an instance of this idea. Further, as AHEAD treats and refines all artifacts in the same way, we believe our results on safe composition are applicable to non-code artifacts as well, as we explained in Section 5.3. Demonstrating this is a subject of future work.

We analyze feature source code to expose properties that must be satisfied by a program in which a feature module can appear. Krishnamurthi and Fisler analyze feature/aspect modules that contain fragments of state machines, and use the information collected for compositional verification [32][33].

Our work is related to *module interconnection languages (MILs)* [18][44] and *architecture description languages (ADLs)* [46] that verify constraints on module compositions. When feature modules are used, a feature model becomes an MIL or ADL.

Our approach to compile individual feature modules and to use bytecode composition tools follows the lead of Hyper/J [40]. However, our technique for compiling feature modules is provisional. A more general approach, one that encodes a language's type theory as module composition constraints, is exemplified by work on separate class compilation [2]. Recent programming languages that support mixin-like constructs, e.g., Scala [39] and CaesarJ [4], suggest an alternative approach to defining, compiling, and composing feature modules. Interestingly, the basic idea is to define features so that their dependencies on other features is expressed via an inheritance hierarchy. That is, if feature `F` extends definitions of `G`, `F` is a "sub-feature" of feature `G` in an inheritance hierarchy. Neither Scala or CaesarJ use feature models, which we use to encode this information. At feature composition time, a topological sort of dependencies among referenced features is performed, which linearizes their composition. The linearization of features is precisely what our feature models provide. One of the advantages that feature models offer, which is a capability that is not evident in Scala and CaesarJ, is the ability to swap features or combinations of features. To us, as long as grammar and cross-tree constraints are satisfied, any composition of features is legal. It is not clear if Scala and CaesarJ have this same flexibility. We believe our work may be relevant to these languages when safe composition properties need to be verified in product line implementations.

Propagating feature selections in a feature model into other development artifacts (requirements, architecture, code modules, test cases, documentation, etc.) is a key problem in product lines [42]. Our work solves an instance of this problem. More generally, verifying properties of different models (e.g., feature models and code implementations of features) is an example of *Model Driven Design (MDD)* [48][34][23][12]. Different views or models of a program are created; interpreters extract information from multiple models to synthesize target code. Other MDD tools verify the consistency of different program (model) specifications. Our work is an example of the latter.

We mentioned earlier that aspects can be used to implement refinements. AHEAD uses a small subset of the capabilities of AspectJ. In particular, AHEAD method refinements are around advice with execution pointcuts that capture a single joinpoint. Aspect implementations of product lines is a topic of current research (e.g., [1][14]), but examples that synthesize large programs or product lines are not yet common. Never the less, the techniques that we outlined in this paper should be relevant to such work.

8 CONCLUSIONS

The importance of product lines in software development will progressively increase. Successful products spawn variations that often lead to the creation of product lines [41]. Coupled with this is the desire to build systems compositionally, and to guarantee properties of composed systems. A confluence of these research goals occurs when modules implement features and programs of a product line are synthesized by composing feature modules.

We examined safe composition properties in this paper, which ensure that there is an absence of references to undefined elements (classes, methods, variables) in a composed program's implementation for all programs in a software product line. We mapped feature models to propositional formulas, and analyzed feature modules to identify their dependencies with other modules. Not only did our analysis identify previously unknown errors in existing product lines, it provided insight into how to create better designs and how to avoid designs that "smell bad". Further, the performance of using SAT solvers to prove theorems was encouraging: non-trivial product lines of programs of respectable size (e.g., product lines with over 50 members, each program of size 35K LOC) could be analyzed and verified in less than 30 seconds. For this reason, we feel the techniques presented are practical.

Our work is but a first step toward more general and useful analyses directed at software product lines. We believe this will be an important and fruitful area for future research.

Acknowledgements. This work was support in part by NSF's Science of Design Project #CCF-0438786. We thank William Cook, David Kitchin, and the referees for their helpful comments.

9 REFERENCES

[1] M. Anastasopoulos and D. Muthig. "An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology". *ICSR 2004*.

[2] D. Ancona, et al. "True Separate Compilation of Java Classes", *PPDP 2002*.

[3] Apache Ant Project. `http://ant.apache.org/`

[4] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. "An Overview of CaesarJ", to appear *Journal of Aspect Oriented Development*, 2005.

[5] Argonne National Laboratory. "Otter: An Automated Deduction System", `www-unix.mcs.anl.gov/AR/otter/`

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

[7] D. Batory, B. Lofaso, and Y. Smaragdakis. “JTS: Tools for Implementing Domain-Specific Languages”. *5th Int. Conference on Software Reuse*, Victoria, Canada, June 1998.

[8] D. Batory, Rich Cardone, and Y. Smaragdakis. “Object-Oriented Frameworks and Product Lines”. *Software Product Line Conference (SPLC)*, August 2000.

[9] D. Batory, AHEAD Tool Suite. **www.cs.utexas.edu/users/schwartz/ATS.html**.

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

[11] D. Batory. “Feature Models, Grammars, and Propositional Formulas”, *Software Product Line Conference (SPLC)*, September 2005.

[12] D. Batory “Multi-Level Models in Model Driven Development, Product-Lines, and Metaprogramming”, *IBM Systems Journal*, Vol. 45#3, 2006.

[13] P. Clements. private correspondence 2005.

[14] A. Colyer, A. Rashid, G. Blair. “On the Separation of Concerns in Program Families”. Technical Report COMP-001-2004, Lancaster University, 2004.

[15] T.H. Cormen, C.E. Leiserson, and R.L.Rivest. *Introduction to Algorithms*, MIT Press,1990.

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

[17] K. Czarnecki and K. Pietroszek. “Verifying Feature-Based Model Templates Against Well-Formed OCL Constraints”, submitted 2005.

[18] T.R. Dean and D.A. Lamb. “A Theory Model Core for Module Interconnection Languages”. *Conf. Centre For Advanced Studies on Collaborative Research*, 1994.

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

[20] M. Flatt, S. Krishnamurthi, and M. Felleisen, “A Programmer's Reduction Semantics for Classes and Mixins”. *Formal Syntax and Semantics of Java*, chapter 7, pages 241--269. Springer-Verlag, 1999.

[21] K.D. Forbus and J. de Kleer, *Building Problem Solvers*, MIT Press 1993.

[22] I. Godil and H.-A. Jacobsen, “Horizontal Decomposition of Prevayler”. *CASCON 2005*.

[23] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories: Assembling Applications with Patterns, models, Frameworks and Tools*, Wiley, 2004.

[24] I.M. Holland. “Specifying Reusable Components Using Contracts”. *ECOOP 1992*.

[25] S.S. Huang, D. Zook, and Y. Smaragdakis. “Statically Safe Program Generation with SafeGen”, *GPCE 2005*.

[26] A. Igarashi, B. Pierce, and P. Wadler, “Featherweight Java A Minimal Core Calculus for Java and GJ”, *OOPSLA 1999*.

[27] M. de Jong and J. Visser. “Grammars as Feature Diagrams”. **www.cs.uu.nl/wiki/Merijn/PaperGrammarsAsFeatureDiagrams**, 2002.

[28] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. “Feature-Oriented Domain Analysis (FODA) Feasibility Study”. Technical Report, CMU/SEI-90TR-21, Nov. 1990.

[29] K. Kang. private communication, 2005.

[30] G. Kniesel, “Type-Safe Delegation for Run-Time Component Adaptation”, *ECOOP 1999*.

[31] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. “Hygienic Macro Expansion”. *SIGPLAN ‘86 ACM Conference on Lisp and Functional Programming*, 151-161.

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

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

[34] V. Kulkarni, S. Reddy. “Separation of Concerns in Model-Driven Development”, *IEEE Software* 2003.

[35] R.E. Lopez-Herrejon and D. Batory. “A Standard Problem for Evaluating Product Line Methodologies”, *GCSE 2001*, September 9-13, 2001 Messe Erfurt, Erfurt, Germany.

[36] R.E. Lopez-Herrejon and D. Batory. “Using Hyper/J to implement Product Lines: A Case Study”, Dept. Computer Sciences, Univ. Texas at Austin, 2002.

[37] J. Liu, D. Batory, and C. Lengauer, “Feature Oriented Refactoring of Legacy Applications”, *ICSE 2006*, Shanghai, China.

[38] C. McManus, The Basics of Java Class Loaders, **www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html**

[39] M. Odersky, et al. An Overview of the Scala Programming Language. September (2004), **scala.epfl.ch**

[40] H. Ossher and P. Tarr. “Multi-dimensional Separation of Concerns and the Hyperspace Approach.” In *Software Architectures and Component Technology*, Kluwer, 2002.

[41] D.L. Parnas, “On the Design and Development of Program Families”, *IEEE TSE*, March 1976.

[42] K. Pohl, G. Bockle, and F v.d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer 2005.

[43] Prevaler Project. **www.prevayler.org/**.

[44] R. Prieto-Diaz and J. Neighbors. “Module Interconnection Languages”. *Journal of Systems and Software* 1986.

[45] SAT4J Satisfiability Solver, **www.sat4j.org/**

[46] M. Shaw and D. Garlan. *Perspective on an Emerging Discipline: Software Architecture.* Prentice Hall, 1996.

[47] Specware. **www.specware.org.**

[48] J. Sztipanovits and G. Karsai. “Model Integrated Computing”. *IEEE Computer*, April 1997.

[49] W. Taha and T. Sheard. “Multi-Stage Programming with Explicit Annotations”, *Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 1997.

[50] M. VanHilst and D. Notkin. “Using C++ Templates to Implement Role-Based Designs”, *JSSST Int. Symp. on Object Technologies for Advanced Software*. Springer Verlag, 1996.

[51] E. Wagner. “Algebraic Specifications: Some Old History and New Thoughts”, *Nordic Journal of Computing*, Vol #9, Issue #4, 2002.

[52] N. Wirth. “Program Development by Stepwise Refinement”, *CACM* 14 #4, 221-227, 1971.