Safe Composition of Product Lines

Sahil Thaker, Don Batory, David Kitchin, and William Cook
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{sahilt,batory,dkitchin,wcook}@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 all programs in a product line are type safe: i.e., absent of references to undefined elements (such as classes, methods, and variables). We show how safe composition properties 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, product lines, 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 to users in an easy-to-understand way, they express functionalities concisely, and help delineate commonalities and variabilities in a domain [30].

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 for 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 [29]. 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 type safe: i.e., absent of references to undefined classes, methods, and variables. Safe composition is related to safe generation and verifying properties of feature-based templates: i.e., providing guarantees that generators synthesize programs with particular properties [48][54] [51] [26][34][17].

In this paper, we show how type safety properties of AHEAD product lines can be verified using feature models and SAT solvers. We identify properties and verify that they hold for all product line members for several product lines. Some properties that we analyze do not reveal actual errors, but rather designs that "smell bad" and that could be improved [22].

2. Formal Models of Product Lines
A *feature model* is a hierarchy of features that is used to distinguish products of a product line [29][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.

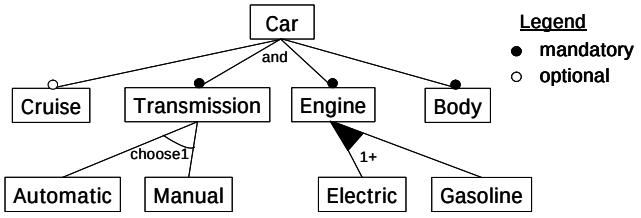


Figure 1. A Feature Diagram

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 [28]. Rules for translating feature diagrams to grammars are listed the **grammar** column of 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 products can have at most one copy of a feature and the order in which features appear in a sentence is the order in which they are listed in the grammar [11].

A specification of a feature model is a grammar and its cross-tree constraints. A model of our automotive product line is listed in

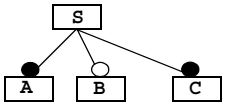
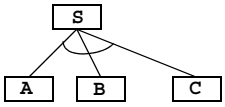
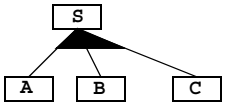
concept	diagram notation	grammar	propositional formula
and		`S : A [B] C ;`	$(S \Leftrightarrow A) \wedge (B \Rightarrow S) \wedge (C \Leftrightarrow S)$
alternative (choose1)		`... S ...` `S : A | B | C ;`	$(S \Leftrightarrow A \vee B \vee C) \wedge$ atmost1(A,B,C)
or (choose 1+)		`... S+ ...` `S : A | B | C ;`	$S \Leftrightarrow A \vee B \vee C$

Figure 2. Feature Diagrams, Grammars, and Propositional Formulas

```
// 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

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].

Feature models are compact representations of propositional formulas [11]. Rules for translating grammar productions into formulas are listed **propositional formula** column of 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 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). Ordering constraints of feature models can also be mapped to formulas, but we choose not to do so for simplicity. It is the translation of feature models into propositional formulas that we will exploit in 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*, representing 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):

```
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 [55]: 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 a 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 implement program refinements. For example, let `BASE` be a feature that encapsulates 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 [10]. Other implementations, such as using aspects, are discussed in Section 6.

3.3 Safe Composition

The problem of safe composition is illustrated by the following example. Let **PL** be a product line with three features: **base**, **addD**, and **refC**. Figure 5 shows their modules. The **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 5. 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.

Currently, the only way to verify that all programs of a product line are type safe is to generate them all and compile them. But product lines can have huge numbers of programs. Brute force is impractical. We need efficient ways to verify type safety (i.e., safe composition) properties of all programs of a product line. In the following sections, we explain how this can be done.

4. Safe Composition Verification

Our solution to safe composition is accomplished in two steps. The first compiles each feature module in order to satisfy lightweight global consistency constraints. The second and more difficult step addresses the combinatorics of feature modules.

4.1 Step 1: Lightweight Global Consistency

The first step in testing safe composition properties is a lightweight global consistency check that compiles all feature modules. This check (a) determines how each class, method, and field reference in every module binds to a declaration, and (b) 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) [41]. 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 exploit the fact that we have the source for all feature modules.[1] For every class, we automatically create a stub that contains the union of the signatures of all fields, methods, and declarations that could appear in that class. This provides us the stubs that are needed to individually compile feature modules, without dealing with the feature combinatorics that is the subject of the next section. An example of a global consistency problem 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 consistency check catches. Consider the base program **BaseP** in Figure 6a, which consists of two interfaces (**I**,**J**) and three classes (**X**,**A**,**B**) **BaseP** is consistent in isolation: the **foo(x)** call in Figure 6a binds to the **foo(I)** method of class **A**.

1. If we had only binaries for some features, we could extract information for our analysis using reflection.

Now consider feature module `ExtendX` of Figure 6b 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`. These checks are verified by the Java compiler when feature modules are compiled individually.

```
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 6. Uncompilable Feature Modules

The next section assumes we have the bytecodes of each feature module from which we can extract field, method, and class and interface references.

4.2 Step 2: Feature Module Combinatorics

The feature model of a product line (Section 2) defines the permitted compositions of feature modules. We want to know if any of these compositions yield programs that are not type safe. This involves examining compositions of feature modules and verifying properties of these compositions. In this section, we identify a set of properties (constraints) that are central to type safety and explain how these properties are verified.

4.2.1 Verification Properties (Constraints)

We identify five properties (constraints) `(2)-(6)` that are essential to safe composition. In Section 6 we consider whether these constraints are sufficient.

Refinement Constraint. Suppose a member or class `m` is introduced (i.e., defined) 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 being refined), and
> (ii) `X`, `Y`, or `Z` must appear in every product that contains feature `F`.

Property (i) is verified by examining the feature model, as it linearizes the composition of features. Property (ii) requires the following constraint to be satisifed:

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

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`.

Let H_n be a superclass of `C`, where `n` is the number of ancestors above `C`. Thus H_0 denotes 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 class `C` or in some superclass of `C` is:

$$F \Rightarrow Sup_0(m) \vee Sup_1(m) \vee Sup_2(m) \vee \ldots \qquad (3)$$

> Note: Special cases of `m` are constructor, field, and super references.

Stepwise refinement requires `F` to be composed after all features that are listed to the right of the implication sign (i.e., `m` must be defined before it is referenced). We examine the feature model to verify this ordering.

Single Introduction Constraint. As a general rule, a member or class is introduced (i.e., defined) only once. If it is defined multiple times, only the last definition is retained. We call this definition *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 (extends) — 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 Members

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 [32]: 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" [22]. We note that this problem was first recognized by Flatt et al in mixin compositions [19], and has resurfaced elsewhere in object delegation [31] 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 (4)
```

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`. 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:

$$F \wedge X \Rightarrow Sup_0(C.m) \vee Sup_1(C.m) \vee Sup_2(C.m) \vee \ldots \quad (5)$$

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 `(5)` 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`. `(5)` would have to be modified to take this possibility into account.

Interface Constraint. Let feature `F` define member `m` in interface `I`. 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 both present in a program, then class `C` must implement or inherit `m`. Let `C.m` denote method `m` of class `C`. The constraint to verify is:

$$F \wedge X \Rightarrow Sup_0(C.m) \vee Sup_1(C.m) \vee Sup_2(C.m) \vee \ldots \quad (6)$$

This constraint is identical in form to `(5)`, although the parameters `F`, `X`, and `m` may assume different values. `F` and `X` must be composed after all features that are listed to the right of the implication otherwise a back-reference violates ordering constraints.[2]

4.2.2 Property Verification

By examining the code base of feature modules, it is possible to identify and collect instances of each of the constraints of Section 4.2.1. These constraints, called *implementation constraints*, are a consequence of feature implementations. They can be added as cross-tree constraints to the feature model and obeying these additional constraints will guarantee type safety. That is, only programs that satisfy domain *and* implementation constraints will be synthesized. Of course, the number of implementation constraint instances may be huge for large programs.

Czarnecki [17] observed that implementation constraints should be implied by domain constraints. 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 ($PL_f \wedge \neg R$) or equivalently $\neg(PL_f \Rightarrow R)$ 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 returned by the solver identifies the offending product. In this manner, we can verify that all products of `PL` satisfy `(2)`. Theorem provers, such as Otter [5], could be used.

> Note: In effect, we are inferring composition constraints for each feature module; these constraints lie at the module's "requires-and-provides interface" [18]. When feature modules are composed, we must verify that their "interface" constraints are satisfied. If composition is a linking process, we are guaranteeing that there will be no linking errors. The difference with normal linking is that we check all combinations of linkings allowed by the feature model.

4.3 Beyond Code Artifacts

The ideas of safe composition transcend source code [17][52]. 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 just outlined could be used to verify safe composition properties of a product line of XML documents (see [52] for an example). The same holds for product lines of other artifacts (grammars, makefiles, etc.) as well. The reason is that we are analyzing properties of structures that are common to all kinds of documents; herein lies the generality and power of our approach.

5. Results

We analyzed the safe composition properties of many different AHEAD product lines. Table 1 summarizes their size statistics. For lack of space in this paper, we explain in detail our findings of the first two product lines listed, and give statistics for the remaining. Note that the size of the code base 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 `(3)`
- Single Introduction `(4)`
- Abstract Class `(5)`
- Interface `(6)`

For each constraint, we generate a proposition to verify that all products in a product line satisfy that constraint. Duplicate propositions can be generated. Consider features `Y` and `ExtendY` of Figure 8. Method `m` in `ExtendY` references method `o` in `Y`, method `p` in `ExtendY` references field `i` in `Y`, and method `p` in `ExtendY` refines method `p` defined in `Y`. We create a proposition for each constraint; all propositions are of the form `ExtendY`⇒`Y`. We elimi-

2. Not all clauses on the right of the implication will result in a feature. For instance, method `C.m` may not be defined at the first level of super-class. Thus, all resulting features on the right hand side of the implication are ones introducing method `C.m`.

```
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`

nate duplicate propositions, and report in our experiments only the number of failures per category. If a proposition fails, our tools report all (in Figure 8, all three) sources of errors. Finally, very few abstract methods and interfaces were used in the product lines of Table 1. So the numbers that we will report in our experiments for 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 [46].

5.1 The 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 [44]. 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.[3]

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

3. We presented a different feature refactoring of Prevayler in [38]. The refactoring we report here is similar to an aspect refactoring of Godil and Jacobsen [23].

Constraint	# of Propositions	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

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

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 other failure was similar. A field member that only the `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.

5.2 The Bali Product Line

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

Constraint	# of Propositions	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 propositions 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:

```
bali2jak ⇒ (require ⇔ requireBali2jak); // new 1
bali2jcc ⇒ (require ⇔ requireBali2jcc); // new 2
composer ⇒ (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 replaces 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 replace `usage()`, rather than refine it with a tool-specific definition. In another case, the replacing 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.

5.3 Other Product Lines

We 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, which incidentally is less time that it takes to generate and compile one of these programs.

6. Future Work

The system and implementation described here indicates an important new direction for ensuring verifiable properties — in particular, safe composition properties — of software product lines. (See [52] for other properties that can be verified).

Although we have implemented this system and evaluated it in practice, we have not presented a formal proof of its correctness. There is as yet no guarantee that all constraints are accounted for. Such a proof does not fit the standard form for proving type soundness. We are developing a new approach that uses a constraint-based type system to account for variations in a product family, and then validating that the constraints are sufficient to ensure safe composition.

We are exploring a formal model of feature composition in a subset of Java. Candidates include *Featherweight Java (FJ)* [27],

Lightweight Java [49], and Classic Java [20]. Roughly, this approach requires four steps:

- Extend the chosen formal model with a syntax for features,
- Define a function that composes features to generate code in the formal model,
- Define a constraint system to validate both global constraints and compositional constraints, and
- Prove formally that this constraint system will only permit the generation of type-safe code.

We have determined that FJ is not suitable as the underlying formal model. This is because FJ encodes a stateless subset of Java, which causes difficulties for features that add fields to an object. FJ requires all fields to be listed in a call to a class constructor, so adding a new field invalidates all existing constructor calls. AHEAD avoids this problem by using Java's default constructors and default values for new fields. We are currently applying our constraint-based approach to Lightweight Java.

7. Related Work

Undefined methods and classes can arise in the linking or run-time loading of programs when required library modules cannot be found [39]. 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 [26], the problem is well-known. The pioneering work of Goguen, Wagner, et al using algebraic specifications to create programs [54], and the work at Kestrel [48] 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 [51]. Huang, Zook, and Smaragdakis [26] 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 closest research to ours, and the 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, our results on safe composition are applicable to non-code artifacts as well. See [52] for an example.

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 [33][34].

Our work is related to *module interconnection languages (MILs)* [18][45] and *architecture description languages (ADLs)* [47] 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 [41]. 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 [40] and CaesarJ [4], suggest an alternative approach to defining, compiling, and composing feature modules. 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 "subfeature" 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 and to define arbitrary propositional constraints for using feature modules. 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 [43]. 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)* [50][35][24][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 join point. 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 [42]. 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 presented techniques for verifying safe composition (i.e., type safety) properties for all programs in a product line. We mapped feature models to propositional formulas, and analyzed feature modules to identify their dependencies on 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 verify propositions 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. Further, our techniques are not limited to composing source code; our analysis is structural, meaning that it can be applied to the synthesis of other non-code artifacts (e.g., XML documents). 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 Shriram Krishnamurthi 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", *Journal of Aspect Oriented Development*, 2006.

[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. "Verification of Feature-Based Model Templates Against Well-Formedness OCL Constraints". *GPCE 2006*.

[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] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[23] I. Godil and H.-A. Jacobsen, "Horizontal Decomposition of Prevayler". *CASCON 2005*.

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

[25] I.M. Holland. "Specifying Reusable Components Using Contracts". *ECOOP 1992*.

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

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

[28] M. de Jong and J. Visser. "Grammars as Feature Diagrams". `www.cs.uu.nl/wiki/Merijn/PaperGrammarsAsFeatureDiagrams`, 2002.

[29] 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.

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

[31] G. Kniesel, "Type-Safe Delegation for Run-Time Component Adaptation", *ECOOP 1999*.

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

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

[34] S. Krishnamurthi, K. Fisler, and M. Greenberg. "Verifying Aspect Advice Modularly", *ACM SIGSOFT 2004*.

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

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

[37] 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.

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

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

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

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

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

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

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

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

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

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

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

[49] R. Strnisza and M. Parkinson, “Lightweight Java: A Fully-Formalized, Extensible Minimal Imperative Fragment of Java”, **http://www.cl.cam.ac.uk/~rs456/lj/**

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

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

[52] S. Thaker. “Design and Analysis of Multidimensional Program Structures”, M.Sc. Thesis, Department of Computer Sciences, The University of Texas at Austin, 2006.

[53] 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.

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

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