Design and Analysis of Multidimensional Program Structures

by

Sahil Thaker, B.E.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

December 2006

Design and Analysis of Multidimensional Program Structures

APPROVED BY
SUPERVISING COMMITTEE:

Don Batory, Supervisor

Dewayne Perry

To my family

Acknowledgments

Seldom is any worthy undertaking tackled alone, and this is no exception. Throughout my stay at the University of Texas many people have influenced my thoughts, provided assistance, guided my work, and comforted me. Listing them here immortalizes their contribution towards my thesis.

First and foremost, my advisor Don Batory must be thanked. His thoughts have influenced mine the most when it comes to Software Engineering. This thesis is primarily a result of his support, guidance, and encouragement throughout the last two years. It reflects his hard-work at least as much as mine.

I also thank Dewayne Perry for reviewing this thesis and sharing his views on Software Engineering. His class on Software Architecture and Design Intent revealed a surprising connection between his work and mine.

Many teachers and colleagues have influenced and shaped my thoughts on Software Engineering and Computer Science. Professors Brown, Novak, Cook, Witchel, and Mooney have had influences on my view which may reflect in this thesis. I also thank colleagues Jia Liu, Sven Apel, Salvador Trujillo, and Mark Grechanik for many fruitful discussions.

Several colleagues have also contributed to the development of tools that I have used. Yancong Zhou must be thanked for writing the ClassReader library and byte-code compilation tools, without which the implementation of safe composition tests would have been immensely harder. Salvador Trujillo built the XML composition tools and helped build tools for XML safe composition. Jia Liu contributed to Prevayler's safe composition analysis.

I am also fortunate for having many other friends and colleagues who have enriched my stay at UT. Their names go unmentioned, but their memories will stay with me forever.

My deepest gratitude goes to my family for unconditional support, love, and inspiration. Every step in my life is influenced by a past that involves them. My father, mother, and sister, in particular, have given me a unique perspective in life. A special thanks must go to my sister, without her annoyance I would have finished writing this thesis weeks earlier. My most ardent gratitude is reserved for Pooravi, my wife-to-be. Her presence has transformed my structural decomposition into another dimension. She has endured as well as accomplished at least as much as I did within the last two years. The support, care, and company that I received from her can neither be replaced nor forgotten. You are an inspiration to me.

SAHIL THAKER

Austin, Texas
November 2006

Design and Analysis of Multidimensional Program Structures

by

Sahil Thaker, M. A.

The University of Texas at Austin, 2006

SUPERVISOR: Don Batory

Software design is an art. More often than not, designers rely on craftsmanship and creativity for designing software products. Just as understanding of the structure of atomic elements is critical to the natural sciences, it too is fundamental to the science behind software engineering. Structural theories simplify our understanding of a convoluted mechanism and allow us to reason in a clear and concise manner. Therefore, decision making can transcend subjective justification and move to quantitative and precise reasoning.

Feature-Oriented Programming and compositional programming embody ideas of software decomposition structure. Feature-Oriented Programming treats Features as the building-blocks that are composed to synthesize programs. Its convergence with algebraic transformations has led to AHEAD - a simple algebraic model of Feature-Oriented Programming. Within AHEAD, reasoning over simple algebraic structures enables codification and mechanization of well-understood principles.

In this thesis we demonstrate just that by showing that critical properties of a decomposition can be identified and verified efficiently. First, we introduce multidimensional structures, a higher-order structural relationship within conventional decomposition. Multidimensional structures simplify a complex decomposition's representation, understanding, and compositional specification. However, not all decompositions are fit for a multidimensional arrangement. We identify an essential property of *orthogonality*

that a decomposition and its implementation must satisfy in order to warrant a multidimensional structure.

Next, we identify a class of errors that occur at module composition-time. These composition errors are a result of unsatisfied dependencies when feature modules are composed in arbitrary arrangements. AHEAD introduces architectural-level domain constraints that govern the compatibility of feature modules.

Besides architectural-level composition constraints, there are also low-level implementation constraints: a feature module can reference classes that are defined in other feature modules. *Safe composition* is the guarantee that programs composed from feature modules are absent of references to undefined classes, methods, and variables. We show how safe composition can be guaranteed for AHEAD product lines using feature models and SAT solvers.

The significance of this work lies in the generality of our approach. Multidimensional structures are not unique to AHEAD; we demonstrate its applicability under any form of decomposition. Likewise, safe composition also extends AHEAD. We draw an analogy between safe composition and C++ linking errors. Moreover, we demonstrate how non-code artifacts benefit from modularization as well as safe composition analysis.

In identifying and verifying key properties, we have also demonstrated how the structural model of AHEAD can simplify our understanding and applicability of essential principles. When the fundamental structure is well-understood, software design is amenable to automation.

Contents

Chapter 1

Introduction

"Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away."
-- *Antoine de Saint-Exupéry*

1.1 Overview

Structural theories are as elemental to program design as they are to the sciences and mathematics. A structural theory is a fundamental way of thinking about relationships within elements of a complex system; without which the simplicity and elegance of description is lost. Principles of large-scale program design may also assume a mathematical form that simplifies their understanding and provides a disciplined way to think about programs, their structure, and their supporting tools. In essence, simplicity of the approach reduces complexity in software development. Such is an ambition of Software Engineering.

Higher-level structured programming was one of the key innovations in reducing complexity from lower-level program description. These ideas have been improved over time, establishing abstraction and modularization as the pillars of structured programming – the foundation of modern Software Engineering. We conjecture that the foundations of tomorrow's technology will not only leverage on these well established ideas, it will have to scale to orders-of-magnitude larger programs.

The contribution of this thesis is to identify further structural relationships that are prevalent within large scale program modularizations and provide the outline of a theory that expresses its structure, manipulation, and synthesis mathematically, and from which properties of a program can be derived from the program's structural description. Such a theory must be defined formally for two reasons: (a) to be able to evaluate its correctness reliably, and (b) to be a reference for an understandable decomposition.

1.2 Problem Description

AHEAD is a model of program synthesis using *Feature-Oriented Programming* and algebraic composition. Its aspiration came from one of the key ambitions of Software Engineering: *to manage complexity as software scales in size*. In particular, AHEAD embodies techniques to build systems compositionally - to break problems into manageable units - and to guarantee properties of composed systems.

Both small and medium-scale programs have been constructed using these techniques. During the process, a key observation was made: *within a hierarchal product line decomposition, there may be further structural relationships that are not best captured by hierarchal modularization*. This idea was later distilled into multidimensional models within AHEAD, but it was not clear what constitutes as a dimension, or why certain decomposition structures are not fit for a multidimensional model.

Furthermore, having a larger number of modules also raises complexity in understanding relationships among them. Subsequently, properties that must hold for a decomposition are neither easy to implement nor to verify, hindering its scalability. In practice, it became increasingly difficult to maintain all modules in a consistent state - in particular, without composition errors. There is a need for theory and tool-support for automated analysis of decomposition properties. This thesis addresses the following questions:

- What properties must a design have to warrant a multidimensional structure?
- What are the properties necessary to ensure a product line lacks composition errors?
- How do we verify these properties?

1.3 Outline

This thesis is structured as follows:

In Chapter 2 we provide the necessary background. We first overview Feature-Oriented Programming, product line engineering, and the fundamentals of AHEAD. Chapter 2 ends with a rigorous treatment of multidimensional models as it forms the basis of our subsequent work.

Next, in Chapter 3 we consider automated analysis of product line models. We show how properties of safe composition can be achieved for AHEAD product lines by using feature models and SAT solvers. We report our findings on several 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.

Multidimensional models are revisited in Chapter 4. We revisit orthogonality and develop an algorithm for verifying orthogonality of multidimensional models. Once again, we provide results of the analysis carried over existing product lines. We also discuss how errors revealed in our analysis can be fixed to make a model orthogonal. Chapter 4 concludes with general perspective of this work and venues for future work.

Chapter 5 overviews the broader picture of our work. It restates the problem, our solutions to those problems, and concludes with the main contributions and significance of this work.

Chapter 2

Background.

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."
-- *Edsger W. Dijkstra*

2.1 Feature Oriented Programming

A software product line is a family of programs where each member differs from others in a well-defined manner. The strength of product line engineering lies in the systematic and efficient creation of products. Features are commonly used to specify and distinguish members of a product line, where a *feature* is an increment in program functionality. Features communicate a product's characteristics clearly; they capture program functionality concisely, and help delineate commonalities and variabilities in a domain [28].

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 key focus in *Feature Oriented Programming (FOP)* is the study of feature modularity in product lines.

Product lines are commonly specified by describing features it may have. Further, 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].

2.2 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 2.1 shows the feature model 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:

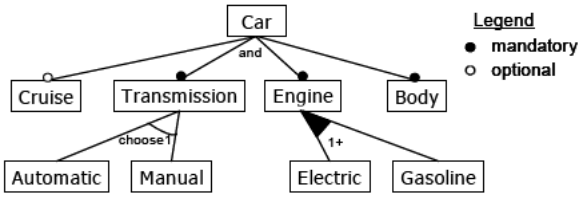


Figure 2.1 A Feature Diagram for a Car Product-Line

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)'*. In the above product line 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 2.2. A bracketed term `[B]` means that feature `B` is optional, and term `S+` means select one or more subfeatures of `S`. We require that subfeature selections are not replicated and the order in which subfeatures

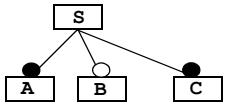
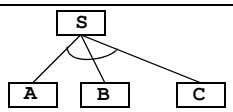
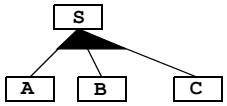
concept	diagram notation	grammar	propositional formula
and		`S : A [B] C ;`	$(S \Leftrightarrow A) \wedge (B \Rightarrow S) \wedge (C \Leftrightarrow S)$
alternative (choose1)		`... S ...` `S : A | B | C ;`	$(S \Leftrightarrow A \vee B \vee C) \wedge atmost1(A,B,C)$
or (choose 1+)		`... S+ ...` `S :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

Figure 2.2 Feature Diagrams, Grammars, and Propositional Formulas

appear in a sentence is the order in which they are listed in the grammar [11]. That is, the grammar enforces composition order.

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 2.3. 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 2.3 A Feature Model Specification

Feature models are compact representations of propositional formulas [11]. Rules for translating grammar productions into formulas are listed in Figure 2.2. (The `atmost1(A,B,C)` predicate in Figure 2.2 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.

2.3 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.

2.3.1 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 fea-

ture 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.

2.3.2 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 2.4a shows the `buffer` class of `BASE` and Figure 2.4b 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 2.4 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 2.4a, a class that is equivalent to that in Figure 2.4b is produced. See [10] for further details.

2.3.3 Principle of Uniformity

A program has many representations beyond source code, including UML documents, process diagrams, makefiles, performance models, and formal specifications, each written in its own language.

AHEAD is based on the Principle of Uniformity [9], meaning that all program representations are subject to refinement. Put another way, when a program is refined, any or all of its representations may change. AHEAD adopts a scalable notion of refinement: impose a class structure on all artifacts, and refine such artifacts similar to code.

Consider an `ant` makefile [63], a typical non-Java artifact. Figure 2.5 shows how a class structure can be imposed on an `ant` artifact: a project is a class, a property is a variable, and a target is a method.

```
<project myMake> ──────────> class myMake {
  <property name="p"  ──────>   p = 1
     value="1"/>
  <target main  ─────────────>  void main() {
    depends="common">             common();
    <compile A />                 compile A;
    <compile B />                 compile B;
    <compile C />                 compile C;
  </target>                     }
  <target common> ────────────> void common() {
    <compile X>                   compile X;
    <compile Y>                   compile Y:
  </target>                     }
</project>                    }

      (a)                           (b)
```

Figure 2.5 Makefile-Class Concept

Refinements of a makefile involve adding new targets and properties (i.e., methods and variables) to a project (class), and extending existing targets (i.e., extending exist-

ing methods). Figure 2.6a shows an `Ant` artifact, Figure 2.6b shows a refinement, and Figure 2.6c shows their composition. Refinements of a makefile involve adding new targets and properties (i.e., methods and variables) to a project (class), and extending existing targets (i.e., extending existing methods).

```
(a) <project name="buffer">
        <property name="back" value="0"/>
        <target name="get">
            <echo message="buf is ${buf}"/>
        </target>
    </project>

(b) <refine name="buffer">
        <target name="restore">
            <antcall target="get"/>
            <property name="buf" value="${back}"/>
            <echo message="buf is ${back}"/>
        </target>
    </project>

(c) <project name="buffer">
        <property name="back" value="0"/>
        <target name="get">
            <echo message="buf is ${buf}"/>
        </target>
        <target name="restore">
            <antcall target="get"/>
            <property name="buf" value="${back}"/>
            <echo message="buf is ${back}"/>
        </target>
    </project>
```

Figure 2.6 Ant Refinement Example

In general, a class structure can be imposed on many different kinds of artifacts: grammar files, expressions, HTML files, performance models, etc. Here we focus only on `Java /Jak` artifacts and `XML-ant` artifacts in verifying properties of the model.

2.4 Multidimensional Designs

As early as high-school we are trained to decompose problems into sub-problems and solve them independently. Not surprisingly, similar techniques have arisen in software design. Given that we have been practicing divide-and-conquer technique, at some point we have all observed that different people decompose a problem in different ways and still come to the same result. This general observation leads us to the idea of Multidimensional Designs.

2.4.1 Orthogonality

Underlying the idea of multidimensional decompositions is a crucial concept of *orthogonality*. Orthogonality in mathematics and sciences pertains to condition of opposition. Lines at right angles are orthogonal. Two vectors are orthogonal if the sum of their inner product is zero - i.e. they balance out. An experimental design with two factors is orthogonal if any effects of one factor balances out effects of the other factor - i.e. sums to zero. Likewise, in software design, the term orthogonality describes also an oppositional relationship: **two abstractions influencing each other, such that composition of either abstraction is identical to the composition of other.** We clarify this idea by example.

Let us define a decomposition for a simple Expression evaluation program. The program deals with simple arithmetic expressions such as `5 + 10`, or `5 - x + 10`, and can perform two operations on any expression - `print` and `evaluate`. We describe two abstractions for the software: a set of Objects - *Add, Subtract, Variable, Constant* - and a set of Operations - *print* and *eval*. Interestingly, we can show a unique relationship between these two abstractions. Consider a matrix representation of this design illustrated in Figure 2.7. Along one dimension operations have been decomposed into Print and Eval, and along the other dimension Objects have been decomposed into Operators and Operands; operators are further decomposed into Add and Subtract, and operands into Variable and Constant, thus forming a hierarchy of decomposition.

			Operations	
			Print	Eval
Objects	Operators	Add	*PrintAdd*	*EvalAdd*
		Sub	*PrintSub*	*EvalSub*
	Operands	Variable	*PrintVar*	*EvalVar*
		Constant	*PrintConst*	*EvalConst*

Figure 2.7 Expressions Matrix

What is unique about this matrix? Both abstractions - objects and operations - influence each other. The *Add* object influences operations *print* and *eval* since printing and evaluating addition operator must be implemented by the program. Conversely, the

Print operation influences all objects since functionality to print each type of operator and operand must be implemented. The same holds for other units of either dimension.

A matrix-entry is a module implementing functionality represented by its co-ordinate within the matrix. Since *Add* operator influences *Print* operation, and vice versa, *PrintAdd*, carries this very functionality - it prints the *Add* node. *EvalAdd*, lying at the intersection of *Add* and *Eval*, evaluates the *Add* node. Likewise, all matrix-entries have a responsibility dependent upon its position within the matrix.

If the matrix represented by this expression program is orthogonal, composing a program through the Operations abstraction - i.e. print (of add, sub, var, const) and eval (of add, sub, var, const) should be identical to composing the program through the Objects abstraction - i.e. Add, Sub, Variable, Const. It follows that under the orthogonality constraint we would obtain an identical program regardless of the order in which abstractions were composed. Figure 2.8 clarifies this idea, where + represents the composition operator. Composing Operations should lead to a semantically identical program as if Objects had been composed (Figure 2.8a) - i.e. they balance out each other. By this line of reasoning we can conclude that going one step further to compose the remaining abstraction also results in a semantically identical program (Figure 2.8b).

a)

	Print + Eval
Add	*PrintAdd + EvalAdd*
Sub	*PrintSub + EvalSub*
Variable	*PrintVar + EvalVar*
Constant	*PrintConst + EvalConst*

=

	Print	Eval
Add + Sub + Variable + Constant	*PrintAdd + PrintSub + PrintVar + PrintConst*	*EvalAdd + EvalSub + EvalVar + EvalConst*

b)

	Print + Eval
Add + Sub + Variable + Constant	*PrintAdd + EvalAdd + PrintSub + EvalSub + PrintVar + EvalVar + PrintConst + EvalConst*

=

	Print + Eval
Add + Sub + Variable + Constant	*PrintAdd + PrintSub + PrintVar + PrintConst + EvalAdd + EvalSub + EvalVar + EvalConst*

Figure 2.8 Different Compositions of the Expressions Matrix

Recap: *Two abstractions are orthogonal if i) decompositions of both abstractions influence each other, and ii) composing them in any order should result in an identical program.*

Without satisfying property (i) the decomposition does not warrant a matrix-like structure; without (ii) we can not claim that the two dimensions of the matrix are balanced. Figure 2.9 shows a 2D matrix that may satisfy property (ii), but not property (i). Dimensions F and G do not influence each other; it may well be a single dimensional design. Chapter 4 on orthogonality testing contains examples where property (i) holds, and the product lines have been designed as a multidimensional model, yet property (ii) does not hold.

		Dim. F		
		F1	*F2*	*F3*
Dim G	*G1*	*a*	*b*	*c*
	G2	*d*	-	-
	G3	*e*	-	-

Figure 2.9 A 2D Matrix

When considering more than two abstractions, the orthogonality property can be tested on any combination of 2 dimensions. Given abstractions `{a, b, c,...}`, if we find that *a* is orthogonal to *b*, *b* is orthogonal to *c,* and *a* is orthogonal to *c*, then we construct a 3-dimensional orthogonal matrix of the design with dimensions *a*, *b*, and *c*. In general, for the design to be *n*-dimensional we must have $\binom{n}{2}$ pairs of abstractions that are orthogonal.

Note that Orthogonality is a property of a design, not just AHEAD designs (see Appendix A). Thus, orthogonality property between abstractions does not mandate static-code composition, it can well be observed under dynamic-runtime composition (such as calling a module). It is not surprising that historically researchers have already observed such unique relationships between abstractions where both abstractions influence each other.

During late 80s Tuscany [53] and Perry [54][55][56] had noted that systems may be organized into multiple dimensions. In [54] Perry remarks: "*It is increasingly common that our software systems have multiple dimensions of organization...we have the notion of features in telephone switching systems that often are orthogonal to the decomposition or design structure — that is, multiple components cooperate in implementing some particular behavior*".

His idea of orthogonality between Features and Design Structures is similar to the property we have just described. For instance, he remarks that in a telephone switching system, *features* are orthogonal to *design structures*. Features - such as recovery and logging - influences design structures - such as node, link, and controller - and vice versa. As a matter of fact, our experience has shown that *operations* and *data structure* abstractions are commonly found to be orthogonal in numerous domains.

Furthermore, observe that a problem domain can be decomposed into orthogonal abstractions but its implementation decomposes along only one abstraction. Without the sophisticated compositional tools this is in fact the common practice. The orthogonality property then demands that if the abstractions are orthogonal, it should not matter which of the abstractions is decomposed. Put another way, if a problem is decomposed into orthogonal abstractions, the designer *has a choice* in decomposing along either one of the abstractions.

Reconsider the network program in Perry's work. One could have chosen to decompose along the Features dimension, leaving structures node, link, and controller *tangled* within the feature decomposition. Alternatively, one could have decomposed the problem into design structures, leaving features such as recovery and logging *tangled* tangled within the design structure decomposition. Both decompositions are valid and could implement the same program. In fact, the later alternative is the most common practice in today's Object-Oriented systems.

As an aside, such observations were among the motivations for *Aspect-Oriented Programming (AOP)* [65]. Under the Object-Oriented decomposition certain functionality cuts-across the object decomposition, and we want to be able to decompose it further. AOP, as with FOP, emerges out of a need to modularize in ways besides simply Objects. It is indeed an instance of multidimensional designs, and a part of much bigger picture called Multidimensional Separation of Concerns [66].

2.4.2 Multidimensional Separation of Concerns

Multidimensional abstractions have been explored in the 90s by Tarr, Ossher, and Harrison under the incarnation of *Multi-Dimensional Separation of Concerns (MDSoC)* [66][40]. MDSoC advocates that modularity can be understood as multidimensional spaces, where dimensions represent different ways of decomposing or modularizing the software - by classes, features, aspects, etc. Units along a dimension are particular instances of that dimension's unit of modularity (e.g., classes). Since each dimension modularizes with respect to a unique model of decomposition (e.g. class, feature), units of a dimension may overlap on the same piece of code with multiple units of other dimensions. Thus, for example, partitioning software by features "cross-cuts" a partition by classes, and vice versa. The basic idea is illustrated in Figure 2.10.

Figure 2.10 shows two different modularizations of an Expression Evaluator. The problem embodies set of operations as well as types of data on which operations are executed. For example, type *ModuloInt* represents a result under modulo arithmetic. Under the class decomposition, *Int* cuts-across feature modules *Integer*, *add* and *multiply*. The same idea applies to class *IntModP*.

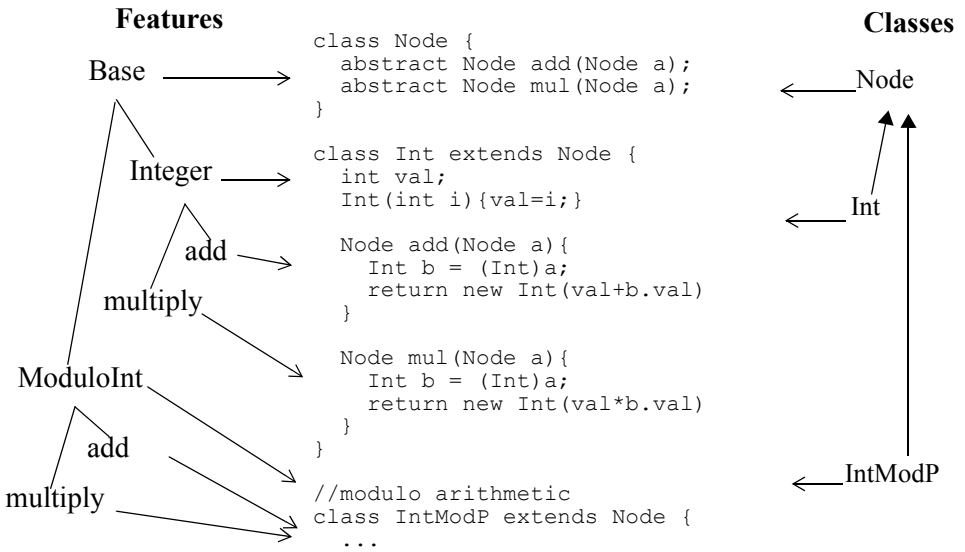


Figure 2.10 Feature and Class decomposition of an Expression Evaluator

In this example Classes are the first manner of decomposition, and Features are secondary. Another way to put it: In AHEAD, features further modularize the collaboration of classes. On the other hand, MDSoC promotes the increasingly popular view that there is no single best manner of decomposition. Accordingly, no preference is given to a particular dimension, all are treated equally.

In contrast to these ideas, this work's contribution is to show that a *single* manner of decomposition also possesses structural relationships that can be classified as multi-dimensional decomposition. Our treatment of multidimensional design explores feature modularity.

Preliminary recognition of *multi-dimensional structures (MDS)* was reported in [67] and [10]. Since then, our understanding of MDS within feature models has evolved considerably. In the subsequent sections we present more general concepts of multi-dimensional structures, and present theory and tools for automating development and analysis of multidimensional designs.

2.5 Multidimensional Designs in AHEAD

Multidimensional feature models are a fundamental design technique in AHEAD. Each dimension provides a unique view of feature decompositions of the program. Unlike other MDSoC approaches, a dimension here implies a unique abstraction of functionality, and *multiple dimensional structures provide multiple feature abstractions over the same implementation.*

A visual depiction of this idea is shown in Figure 2.11. Depending on the task, one may want to view the Feature abstractions, or the Data Structure abstractions. Given these two views we can represent it within a 2-dimensional matrix, where each cell represents a module whose responsibility is denoted by its coordinate. So in the Expressions matrix of Figure 2.7 module *PrintAdd* carries functionality to print out an Addition operator, module *EvalVar* evaluates the value of a variable node, and so forth.

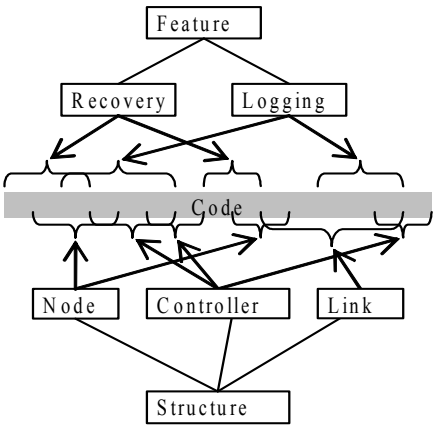


Figure 2.11 Two Views of a Single Code Base

2.5.1 Multidimensional AHEAD Model

To be more precise, consider the AHEAD model `F = {F1, F2, ...F`$_n$`}`. Let program `G = F8+F4+F2+F1` be a program that is synthesized from `F`. We can rewrite `G`'s specification as:

$$G = \Sigma_{i \in (8, 4, 2, 1)} F_i \ ,$$

where `(8,4,2,1)` is the sequence of indices to be summed.

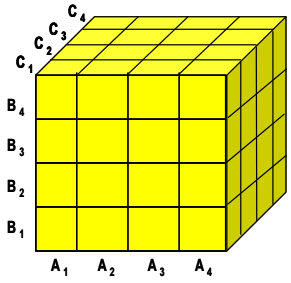


Figure 2.12 A 3D FOP Model

`F` is a 1-dimensional model. An *n*-dimensional model uses *n* FOP models to specify the features (or indices) along a particular dimension. A 3-dimensional model `M` is depicted in Figure 2.12. It uses `A={A`$_1$`, A`$_2$`, ... A`$_a$`}`, `B={B`$_1$`, B`$_2$`, ... B`$_b$`}`, and `C={C`$_1$`, C`$_2$`, ... C`$_c$`}` as dimensional models. The entries of this matrix define model `M={M`$_{111}$`, ... M`$_{abc}$`}`, which has `a*b*c` features, where `M`$_{ijk}$ implements the combined features `(A`$_i$`, B`$_j$`, C`$_k$`)`.

AHEAD models with more than 1 dimensions are called *Multidimensional Models (MDMs)*. In a *n*-dimensional model, a program is specified by *n* equations, one per dimension. For the 3-D model of Figure 2.12, a program `P` in the product-line of `M` would be defined by 3 equations, such as:

$\mathbf{P = A_6 + A_3 + A_1}$
$\mathbf{P = B_7 + B_4 + B_3 + B_2}$
$\mathbf{P = C_9 + C_1}$

This specification is translated into an **M** equation by summing the elements of **M** along each dimension, using the equations above. That is:

$$P = \Sigma_{i \in (6, 3, 1)} \Sigma_{j \in (7, 4, 3, 2)} \Sigma_{k \in (9, 1)} M_{ijk}$$

One of the advantages of multidimensional models is conciseness. Given n dimensions with d features per dimension, program module complexity is $O(d^n)$. However, specification complexity is only $O(dn)$. Thus, *multidimensional models make the application of FOP more scalable as program specifications are exponentially shorter.*

The other advantage is understandability. Since MDMs delineate structural relationships between abstractions, they help us understand how abstractions and modules that implement them are related. See Appendix A for examples.

2.5.2 Properties of AHEAD Models

Multidimensional models rely on the property of orthogonality for consistency. But how do we ensure that an AHEAD model is orthogonal? The first condition, that abstractions (dimensions) should influence each other is a prerequisite for there to be multiple dimensions within the model. How we verify the second condition, that an MDM can be composed along any dimension, is the subject of Chapter 4 on Orthogonality Testing.

The utility of MDMs is that we can specify exponentially large composition of modules through essentially linear length MDM specification. As the number of feature modules increase, it becomes increasingly important to automatically verify certain properties of the model. While dealing with fewer feature modules it may be possible to manually verify or design such properties, but doing so quickly becomes intractable as the number of modules increase. One such property is to ensure that no feature module may be absent of type dependencies - that is, all possible compositions will compile without dependency errors. How we verify this property is the subject of this next chapter on Safe Composition.

Chapter 3

Safe Composition

"I have not failed. I've just found 10,000 ways that won't work."
--*Thomas Edison*

3.1 Overview

The essence of software product lines is the systematic and efficient creation of products. In AHEAD a program is typically specified declaratively 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].

When composing feature modules, a problem that can arise is that there may be references to classes or members that are undefined. The AHEAD tool suite has multiple ways to compose feature modules to build a product. We can compile individual feature

modules and let AHEAD compose their bytecodes to produce the binary of a product directly, or compose the source files and then compile them. Regardless of the approach, it is possible to discover errors (i.e., reference to undefined elements) during the last phase of composition/compilation. This may be too late. In other words, we need to ensure apriori that all variables, methods, and classes that are referenced in a generated 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 3.1 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**; the refinement 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 3.1 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.

3.2 Properties of Safe Composition

3.2.1 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:

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

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, but a majority of implementation 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 in [17] observed the following: Let $\mathbf{PL}_\mathbf{f}$ 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 ($\mathbf{PL}_\mathbf{f} \wedge \neg\mathbf{R}$) 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 ($\mathbf{PL_f} \wedge \mathbf{F} \wedge \neg\mathbf{X} \wedge \neg\mathbf{Y} \wedge \neg\mathbf{Z}$) 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.

3.2.2 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 $\mathbf{H_n}$ be a superclass of **C**, where **n** indicates the position of $\mathbf{H_n}$ by the number of ancestors above **C**. Thus $\mathbf{H_0}$ is class **C**, $\mathbf{H_1}$ is the superclass of **C**, $\mathbf{H_2}$ is the super superclass of **C**, etc. Let $\mathbf{Sup_n(m)}$ denote the predicate that is the disjunction of all features that define method **m** in $\mathbf{H_n}$ (i.e., **m** is defined with a method body and is not abstract). If features **X** and **Y** define **m** in $\mathbf{H_1}$, then $\mathbf{Sup_1(m)=X \vee Y}$. If features **Q** and **R** define **m** in $\mathbf{H_2}$, then $\mathbf{Sup_2(m)=Q \vee R}$. And so on. The constraint that **m** is defined in some superclass is:

$$\mathbf{F} \Rightarrow \mathbf{Sup_1(m)} \vee \mathbf{Sup_2(m)} \vee \mathbf{Sup_3(m)} \vee \ldots \qquad (4)$$

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**.

3.2.3 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:

$$\mathtt{F} \Rightarrow \mathtt{Sup}_0(\mathtt{m}) \vee \mathtt{Sup}_1(\mathtt{m}) \vee \mathtt{Sup}_2(\mathtt{m}) \vee \ldots \quad (5)$$

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:

$$\mathtt{F} \Rightarrow \mathtt{Sup}_0(\mathtt{m}) \quad \texttt{// same as } \mathtt{F} \Rightarrow \mathtt{X} \vee \mathtt{Y} \vee \mathtt{Z} \quad (6)$$

3.2.4 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 3.2a). Feature **B** replaces — not refines — the **get()** method by returning the double of the integer

```
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 3.2 Overriding Members

member (Figure 3.2b). 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 3.2c). If subsequent features depend on the **get()** method of **A**, the resulting program may not work correctly.

It is possible for multiple introductions to be correct; in fact, we carefully used such designs in building AHEAD. More generally, 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**.

3.2.5 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:

$$\mathtt{F} \wedge \mathtt{X} \Rightarrow \mathtt{Sup}_0\mathtt{(C.m)} \vee \mathtt{Sup}_1\mathtt{(C.m)} \vee \mathtt{Sup}_2\mathtt{(C.m)} \vee \ldots \qquad (8)$$

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.

3.2.6 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:

$$\mathbf{F} \wedge \mathbf{X} \Rightarrow \mathbf{Sup}_0\mathbf{(C.m)} \vee \mathbf{Sup}_1\mathbf{(C.m)} \vee \mathbf{Sup}_2\mathbf{(C.m)} \vee \ldots \quad (9)$$

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

3.2.7 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 3.5.1. Further, using a different compilation technology may introduce even more constraints to be checked (see Section 3.5.2).

To determine if we have a full compliment of constraints requires a theoretical result on the soundness of the type system of the Jak langauge. 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 possi-

ble to determine if any constraints are missing for that language subset. To do this for Jak is a topic of future work.

3.2.8 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.

3.3 Generalizing to Multiple Dimensions

One of the motivations for safe composition is to efficiently analyze product lines with large set of feature modules. Multidimensional models succicntly specify a higher-order structure of a large number of possible modules. Therefore we argue that designs that use multidimensional models will benefit the most from safe composition analysis. Our analysis of safe composition thus far has assumed a single-dimensional decomposition. Fortunately it is easy to generalize these ideas to multidimensional model. We demonstrate how below.

Let us first distinguish key differences in specifying a single-dimensional and a multidimensional model. Figure 3.3 shows two product lines consisting of four feature modules `a`, `b`, `c`, and `d`. Both are identical, yet their specification carries different form. A single-dimensional model's grammar and constraints reference feature modules themselves. The Safe composition ideas we have demonstrated so far are over this type of specification. The process is straightforward: for the product line shown in Figure 3.3, analyze

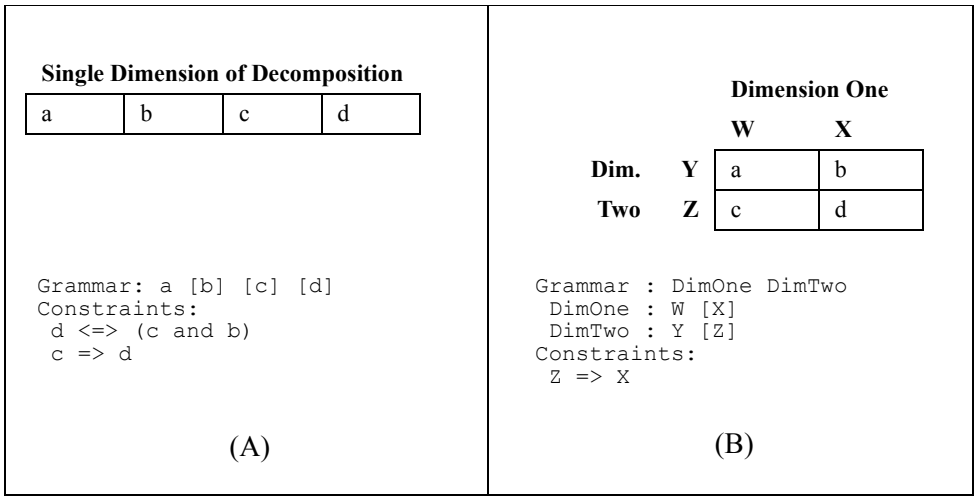


Figure 3.3 An identical one and two dimensional feature model

modules **{a,..,d}**, and append constraints (from Section 3.2) regarding presence of these modules to the grammar that, when satisfied, would describe a safe composition error.

Multidimensional models are specified over dimensions and their units. Individual feature modules are mapped to a coordinate system within those dimensions. For example, to map the model of Figure 3.3A to Figure 3.3B we would construct a mapping as follows:

a --> (W and Y)

b --> (X and Y)

c --> (W and Z)

d --> (X and Z)

The above mapping can be read as: *"Replace* **a** *with* **(W and Y)**, *replace* **b** *with* **(X and Y)**", and so forth. Hence, constraints within Figure 3.3A translate to:

(X and Z) <=> ((W and Z) and (X and Y))

(W and Z) => (X and Z)

Now, **Y** and **W** are always selected in model Figure 3.3b, hence they are always "true". The above generated constraints simplify to:

(X and Z) <=> (Z and X)

```
Z => (X and Z)
```

The first constraint is always true, thus, the multidimensional model of Figure 3.3B carries only the second constraint, which is further simplified to:

```
Z => X
```

Translation of safe composition constraint to a representation suitable for multidimensional models is straightforward. In the case of the model in Figure 3.3B we analyze modules `{a,..,d}` and append constraints describing presence of these modules. The only difference being that in the constraint each module is replaced by conjunction of units where the module is placed. For example, if through code analysis we find that module `c` references a type defined in module `d`, this means `c => d` and we would append the constraint `c and !d`[1] to the product line model. This translates to:

```
(W and Z) and !(X and Z)                                    (10)
```

Now, because `W` can not be absent according to the model in Figure 3.3, and because `Z` requires `X`, conjunction of the original model and (10) can not be satisfied - i.e. there is no assignment that violates the constraint: `c => d`. This particular constraint passes the safe composition test; another may not. We still need to validate *all* dependency constraints.

Safe composition verification is independent of the dimensionality of the model representation since there is direct mapping from the module to its coordinate representation in the model.

3.4 Results

We have analyzed the safe composition properties of many AHEAD product lines. Table 1 summarizes the key size statistics for several of the Java product lines that we analyzed. 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.

1. The constraint that must hold is *c => d*. In order to verify this we append *(c and !d)* to the original model, and let the SAT-solver find an assignment that violates the constraint.

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

TABLE 1. Java Product Line Statistics

The properties that we verified for the Java product lines are grouped into five categories:

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

We also analyzed two XML product lines of **`ant`** files. Ant files also demand product line modularization since one or more features may require the build process to be customized. Hence, a product line which modularizes Java artifacts may also require Ant file modules to be decomposed within its features. Table 2 summarizes statistics for two XML product lines that we analyzed.

Product Line	# of Features	# of Programs	Code Base XML LOC	Program XML LOC
Ahead Tools (ATS)	12	200	3200	400
Pink Creek (PCP)	23	9700	2086	1329

TABLE 2. XML-ant Product Line Statistics

The properties that we verified for XML product lines are grouped into three categories:

- **Refinement** (2)**,**
- **Reference to Member or Class**, includes (4) and (5)
- **Single Introduction** (7)

Abstract classes and Interfaces are not present in the simple mapping from XML structure to the class structure illustrated in Section 2.3.3. Henceforth, we refer only to the

class structure and not the XML structure. Like with Java code, XML code modules are analyzed to extract dependencies in the form of safe composition constraints.

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 Figure 3.4. 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 3.4, 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.

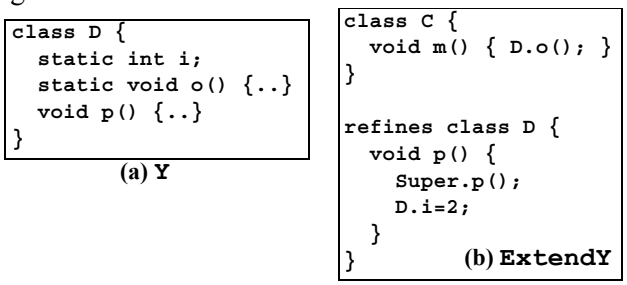


Figure 3.4 Sources of **`ExtendY`**⇒**`Y`**

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].

3.4.1 Graph Product Line

The *Graph Product-Line (GPL)* is a family of graph applications that was inspired by early work on modular software extensibility [19][42]. Each GPL application implements one or more graph algorithms. A feature model for GPL, consisting of 18 distinct features, is listed in Figure 3.5.

A GPL product implements a graph. A graph is either **`Directed`** or **`Undirected`**. Edges can be **`Weighted`** with non-negative integers or **`Unweighted`**. A graph application requires at most one search algorithm: depth-first search (**`DFS`**) or breadth-first search (**`BFS`**), and one or more of the following algorithms:

Vertex Numbering (**`Number`**): A unique number is assigned to each vertex.

Connected Components (`Connected`): Computes the *connected components* of an undirected graph, which are equivalence classes under the reachable-from relation. For every pair of vertices **`x`** and **`y`** in a component, there is a path from **`x`** to **`y`**.

Strongly Connected Components (`StrongC`): Computes the *strongly connected components* of a directed graph, which are equivalence classes under the reachable relation. Vertex **`y`** is reachable from vertex **`x`** if there is a path from **`x`** to **`y`**.

Cycle Checking (`Cycle`): Determines if there are cycles in a graph. A cycle in a directed graph must have at least 2 edges, while in undirected graphs it must have at least 3 edges.

Minimum Spanning Tree (`MSTPrim`, `MSTKruskal`): Computes a *Minimum Spanning Tree (MST)*, which contains all the vertices in the graph such that the sum of the weights of the edges in the tree is minimal.

Single-Source Shortest Path (`Shortest`): Computes the shortest path from a source vertex to all other vertices.

```
// grammar

GPL : Driver Alg+ [Src] [Wgt] Gtp ;
Gtp : Directed | Undirected ;
Wgt : Weighted | Unweighted ;
Src : BFS | DFS ;
Alg : Number | Connected | Transpose StronglyConnected
    | Cycle | MSTPrim | MSTKruskal | Shortest ;
Driver : Prog Benchmark ;

%% // cross-tree constraints

Number ⇒  Src ;
Connected ⇒  Undirected ∧ Src ;
StronglyConnected ⇒  Directed ∧ DFS ;
Cycle ⇒  DFS ;
Shortest ⇒  Directed ∧ Weighted ;
MSTKruskal ∨ MSTPrim ⇒  Undirected ∧ Weighted ;
MSTKruskal ∨ MSTPrim ⇒  ¬(MSTKruskal ∧ MSTPrim) ; // mutual excl.
```

Figure 3.5 Graph Product Line Model

Not all combinations of GPL features are possible. The rules that govern compatibilities are taken directly from algorithm texts and are listed in Figure 3.6, and as cross-tree constraints in Figure 3.5. Note: **`MSTKruskal`** and **`MSTPrim`** are mutually exclusive (the last constraint listed in Figure 3.5); at most one can appear in a GPL product.

Algorithm	Required Graph Type	Required Weight	Required Search
Vertex Numbering	Any	Any	BFS, DFS
Connected Components	Undirected	Any	BFS, DFS
Strongly Connected Components	Directed	Any	DFS
Cycle Checking	Any	Any	DFS
Minimum Spanning Tree	Undirected	Weighted	None
Shortest Path	Directed	Weighted	None

Figure 3.6 Feature Constraints in GPL

The code base of the 18 GPL features is 1800 Jak (or 1800 Java) LOC. Enumerating the GPL product line yields 80 different programs, where a typical program has 5 features and its average source size is 700 Jak (or 700 Java) LOC.

Results. Table 3 summarizes our analysis of GPL. In total, we generated 615 theorems, of which 551 were duplicates. Analyzing the GPL feature module bytecodes, generating theorems and removing duplicates, and running the SAT solver on the remaining 64 theorems took 3 seconds.

Constraint	# of Theorems	Failures
Refinement Constraint	42	0
Reference to Member or a Class	546	0
Introduction Constraint	27	1
Abstract Class Constraint	0	0
Interface Constraint	0	0

TABLE 3. Graph Product Line Statistics

There was one error (or rather a “bad smell” warning), which occurs when the **`Graph.setEdge()`** method is introduced more than once. This “bad smell” stems from the inability of AHEAD to refine the argument list of a method. The base feature of a GPL application is either a **`Directed`** or **`Undirected`** graph. Both features introduce a **`Graph`** class whose add-an-edge method should be:

```
void addEdge( Vertex start, Vertex end );                (11)
```

The **Weighted** feature converts the **Directed** or **Undirected** graph into a weighted graph and the method that adds an edge should be refined with a weight parameter:

```
void addEdge( Vertex start, Vertex end, int weight );    (12)
```

As AHEAD does not support this kind of refinement (and interestingly, neither do the AspectJ or Hyper/J tools), we decided to define **addEdge** with a weight parameter that was ignored in the **Directed** or **Undirected** implementations, but have **addEdge** overridden with an implementation that used the parameter in the **Weight** feature. Every GPL program includes a **Driver** feature that loads and executes the program's algorithms. For **Driver** to be general, it reads files that encode weighted graphs, which means that it always invokes the refined **addEdge** method (12). If the program implements unweighted graphs, the weight parameter is ignored.

Perhaps a cleaner redesign would be to have the **Weight** feature add method (12) and make (11) private, so that (11) could not be used if the **Weight** feature is present in a program. However, this change would complicate the **Driver** feature as it would either invoke (11) or (12). Our analysis revealed this compromise.

3.4.2 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 Prevalyer into the *Prevaler Product Line (PPL)* by giving it a feature-oriented design. That is, we refactored Prevalyer into a set of feature modules, some of which could be removed to produce different versions of Prevalyer 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[2]. 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 3.7. 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 3.7 Prevayler Feature Model

Results. The statistics of our PPL analysis is shown in Table 4. 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 14 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 3.7) we found 11 reference failures, of which 4 were unique errors, and still had 12 multiple-introduction failures. These are the results in Table 4.

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

2. 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].

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 4. Prevayler Statistics

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.

3.4.3 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 3.8. 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 3.8 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 3.8 Bali Feature Model

The statistics of our BPL analysis is shown in Table 5. 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.

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 3.8 allows a Bali tool to have **`requireBali2javacc`** without **`require`**. A similar error was

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 5. Bali Product Line Statistics

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 3.8 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 refining 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.

3.4.4 Jakarta Product Line

The *Jakarta Product Line (JPL)* is a set of AHEAD tools that manipulate and transform Jak files [9]. There are 70 different features that can be composed to build 56 different products. The average number of features per product is 15, and the average size of a JPL program is about 22K Jak LOC or 35K Java LOC that includes a generated parser. The code base is 48K Jak LOC plus grammar files from which parsers can be generated. The statistics of our JPL analysis is shown in Table 6. We generated a 23480 theorems, of which 22987 were duplicates. To analyze the JPL code base, identify and prove the 493 unique theorems took 33 seconds.

Our analysis uncovered multiple errors in JPL. One revealed an unexpected dependency between a feature **`sm5`** that added state machines to the Jak language and a feature **`ast`** that added metaprogramming constructs to Jak [6]. An example of the **`ast`** metaprogramming constructs in Jak is:

```
e = exp{ 3+x }exp;
```

which assigns the parse tree for expression '**`3+x`**' to variable **`e`**. In effect, **`ast`** adds LISP-like quote and unquote to the Jak language. Jak preprocessors replace **`exp{3+x}exp`** with an expression of nested parse tree node constructors. The **`sm5`** feature translates state machine specifications into Java, and uses **`ast`** metaprogramming constructs to express this translation. Compiling a Jak file requires it to be translated first to its Java counterpart, and then **`javac`** is used to compile the Java file. In this translation, calls to methods belonging to the **`ast`** feature appear in the Java expansion of **`ast`** metaprogramming constructs. Although the Jak *source* for **`sm5`** does not reference methods in **`ast`**, its Java *trans-*

Constraint	# of Theorems	Failures
Refinement Constraint	701	0
Reference to Member of a Class	22142	3
Introduction Constraint	247	58
Abstract Class Constraint	358	0
Interface Constraint	32	0

TABLE 6. Jakarta Product Line Statistics

lation does. This implementation dependency was not obvious. We decided that the error is not in our feature implementations, but rather in the JPL feature model: the constraint (the use of the **`sm5`** feature requires the inclusion of the **`ast`** feature) needs to be added.

The most interesting revelation was the discovery of three real errors in JPL feature modules, all having to do with multiple introductions of the same method. Unlike previous errors of this type, the overriding methods were not replicas. Two of the methods returned different String values of method names and method signatures, the difference being that one version trimmed strings of leading and trailing blanks, while the other did not. The third example was less benign, revealing that JPL tools were not processing error messages involving nested interfaces within classes and interfaces correctly.

Our analysis also revealed a situation where a feature introduced several methods in a class, and a subsequent feature removed (stubbed) the functionality of these methods. (The original definition of the methods wrote data to a file; the removal or stubbing eliminated file writing). This is another "bad smell" in JPL's design; a better design would provide one feature for file writing and another for no-op action. By making this functionality explicit, the overall program design becomes clearer.

3.4.5 Ahead Tool Suite

Ahead Tool Suite (ATS) is a collection of tools for Feature-Oriented Programming using the AHEAD model. Bali product line and Java Product Line are two of several tools within this collection. While feature variations of these two tools do not require variations in the build process, variations of tools to be included in ATS (such as include/exclude bali/java) do necessiate changes to the **`Ant`** makefile. The grammar for the ATS product line is shown in Figure 3.9.

- **core** is the base constant feature module
- **reform** is a tool for reformatting jak code with indentation, line-breaks, etc.
- **jedi** is a tool for extending and composing javadoc-like documentation
- **jrename**, **jak2java** are modules required for compilation of jak code

```
// grammar

APL : core Java [Xml] [cpp] [Gui] [drc] [bc] [aj] :: mainAPL ;

Java: [reform] [jedi] [jrename] jak2java bali :: mainJava ;

Xml : xc | xak ;

Gui : [me] [mmatrix] guidsl [web] :: mainGUI ;

bc  : [bcjak2java] :: mainBc ;

aj  : [jak2aj] :: mainAj ;

%% // constraints

bc implies mmatrix ;
me implies mmatrix ;
```

Figure 3.9 ATS Product Line Model

- **bali** is a module for extending and composing BNF form grammar
- **xak** supports giving class structure to XML documents
- **xc** composes xak XML code
- **cpp** is C++ composition tool
- **Gui** modules are GUI tools for building AHEAD applications
- **bcjak2java** provides compiler support for converting bali-jak to java
- **jak2aj** provides compiler support for converting jak code to AspectJ code

The code base of the 12 ATS features is 3200 XML LOC. We discovered three errors in the model. Features *core*, *drc*, and *jrename* were missing from the grammar. Figure 3.9 is a result of fixing these errors.

Table 7 summarizes our analysis of ATS after fixing the model. In total, we generated 829 theorems, of which 807 were duplicates. Analyzing the ATS feature module's XML files, generating theorems and removing duplicates, and running the SAT solver on the remaining 22 theorems took 10 seconds.

Constraint	# of Theorems	Failures per Category
Refinement Constraint	11	0
Reference to Member or a Class	818	0
Introduction Constraint	0	0

TABLE 7. ATS Statistics

3.4.6 Pink Creek Product (PCP)

The PinkCreek product line is a family of web portals. A web portal is a web-site that provides a single point-of-access to the resources available on the internet - such as email, weather, news, etc - and allows for its customization. A web-portlet is a reusable web-component representing one of the services on a portal web-site. The code for a given service may span web user interface, middleware and backend code. Another way to understand portlets is to consider it as a cross-section of a typical multi-tier architecture. A portlet is in essence a feature of the portal web site.

PinkCreek Product is a web-portlet that provides flight reservation capabilities to different web-portals. Its functionality is roughly: (i) search for flights, (ii) present flight options, (iii) select flights, and (iv) purchase tickets. Figure 3.10 shows the AHEAD model of PCP product line.

```
// grammar

FBP : BASE Container [INCLUDE_SOURCE_CODE] [INCLUDE_JAVADOC] ::
MainGpl ;

Container : EXO_TOMCAT | EXO_JBOSS | EXO_JONAS ;

%%

EXO_JBOSS implies EXO_JBOSS ;
```

Figure 3.10 PCP Product Line Model

The code base for build process (makefile) of 23 PCP features is 2086 XML LOC. In total, we generated 89 theorems, of which there were 76 duplicate theorems. Analyzing

PCP feature module's XML files, generating theorems, removing duplicates, and running the SAT solver on the remaining 13 theorems took 2 seconds.

Table 8 summarizes our analysis of PCP Product Line. We encountered 2 failures, one of which was a multiple-introduction warning. The second error revealed an error in our model: feature *Container* was declared optional when it shouldn't be. Figure 3.10 shows our model of PCP after fixing this error.

Constraint	# of Theorems	Failures per Category
Refinement Constraint	1	1
Reference to Member or a Class	84	0
Introduction Constraint	4	1

TABLE 8. PCP Statistics

3.5 Related and Future Work

3.5.1 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 for features to 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 3.11. **`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 3.11c.

Observe that **Ref•Base** (Figure 3.11c) satisfies our safe composition constraints, namely there are 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 3.11 Ordering Example

We can circumvent this problem by rewriting **Base** as **Base1** in Figure 3.12, 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 3.11c. 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 3.12 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.

3.5.2 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 closest 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., include this element 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 3.2.8. Demonstrating this is a subject of future work.

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. 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. 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. In any case, 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 Chapter should be relevant to such work.

3.6 Summary

We examined safe composition properties in this chapter, 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.

Chapter 4

Orthogonality Testing

"Beware of bugs in the above code; I have only proved it correct, not tried it."
-- *Donald E. Knuth*

4.1 Overview

Multidimensional models posit natural occurrence of orthogonal abstractions in program decompositions. While we do not pursue here techniques for deriving such abstractions, we focus solely on verifying an implementation for orthogonality once the right abstractions have been designed.

The idea of orthogonality is of balanced abstractions. That is, two abstractions influence each other to an equal degree, such that either choice of decomposition is yields the same result. With automated composition technologies such as AHEAD it is further possible to simultaneously decompose along multiple abstractions. One of the tenets of this approach is the ability to synthesize software modules that capture different design viewpoints or perspectives.

This rests on a fundamental assumption: *different projections of the code-base should be consistent*. If they are inconsistent, then the abstractions are not defined to be balanced, and we may obtain decompositions that represent different programs. Likewise, any edits that we perform on a view may not correspond to correct functionality in another view. Stated differently, to produce consistent views of a multidimensional code-base and allow programmers to edit these views, the underlying multidimensional model must sat-

isfy the orthogonality property: *the same program will be produced when matrix dimensions are summed in any order*. The orthogonality property is an algebraic property of summation commutativity. For a 2-dimensional model `A`, the orthogonality property is expressed by:

$$\Sigma_i \Sigma_j A_{ij} = \Sigma_j \Sigma_i A_{ij}$$

For *n* dimensions, there are *n!* different summation orders, all of which must produce syntactically equivalent results. In reality, we want to guarantee semantic equivalence. Unfortunately, guaranteeing semantic equivalence is a much harder problem. Syntactic equivalence implies semantic equivalance, and it is a simpler problem to solve.

Given this definition of orthogonality we describe our orthogonality test and an algorithm for testing if a multidimensional model is orthogonal. We outline the basic approach and our results in the following sections.

4.2 Example of a Non-Orthogonal Model

The primary problem we trying to solve is to verify orthogonality of an implementation of a multidimensional model. But first we demonstrate how non-orthogonal abstractions arise. To see this, consider a slightly condensed version of the Expression model from Figure 2.7. For brevity, only *Variable* and *Add* units are present along the *Object* dimension. The revised matrix is shown in Figure 4.1.

		Operations	
		Print	Eval
Objects	Variable	*PrintVar*	*EvalVar*
	Add	*PrintAdd*	*EvalAdd*

Figure 4.1 Smaller Expressions Matrix

One possible implementation of the Expression product line is shown in Figure 4.2. Feature *PrintVar* introduces class *Var* with the method *print* and some data members. It also adds a class *Test*, which creates a *Var* object and prints it. Feature *EvalVar* refines classes *Var* and *Test* introduced in *PrintVar*. It introduces a method to evaluate the

PrintVar

```
class Var{
 public int value;
 public String name;

 print(){
   Sys.println(name+"="+value);
 }
}

class Test{
 main(){
   Var var = new Var();
   var.name="x"; var.value=10;
   var.print();

 }
}
```

EvalVar

```
refines class Var {
  int eval(){
    return value;
  }
}



refines class Test{
  main(){
    Super().main();
    Sys.println(var.eval())
  }
}
```

PrintAdd

```
class Add{
 public Var left;
 public Var right;

 print(){
   Sys.println(
     left.eval()+"+"+right.eval());
 }
}

refines class Test{
 main(){
  Super().m();
  Var left = new Var();
  var right = new Var();
  left.name="a"; right.name="b";
  left.value=2; right.value=3;

  Add add = new Add();
  add.left=left; add.right=right;

  add.print();
 }
}
```

EvalAdd

```
refines class Add {

  int eval(){
    return left.eval()+right.eval();
  }
}



refines class Test{
  main(){
    Super().main();
    Sys.println(add.eval())
  }
}
```

Figure 4.2 Feature Implementation of the Expression Product Line

value of a variable, and appends the test to print out the value of *Var* object created by *PrintVar*.

Feature *PrintAdd* introduces a class *Add*, with data members and print operation. It also refines *Test*, introduced by *PrintVar*, by printing out an *Add* node. *EvalAdd* feature refines *PrintAdd*'s *Add* class by introducing an operation to evaluate the value of an *Add* node, and refines the *Test* class to print out the value of an *Add* node.

This particular implementation does not satisfy the orthogonality property. Aggregating Operations and then Objects produces:

```
PrintVar + EvalVar + PrintAdd + EvalAdd                    (13)
```

Aggregating Objects and then Operations produces:

```
PrintVar + PrintAdd + EvalVar + EvalAdd                    (14)
```

When the program of equation (13) is run it outputs: `x=10, 10, 2+3, 5.` The program of equation (14) outputs: `x=10, 2+3, 10, 5.` Therefore, in this implementation, composition of *EvalVar* and *PrintAdd* does not commute. There are essentially two reasons why it can not be composed in two different orders.

First, *PrintAdd's print* method references *EvalVar's eval* operation. This is permitted in the product line of equation (13) since *PrintAdd* is composed after *EvalVar*. However it fails under the tenet of step-wise refinement when composed using equation (14). A module can not reference a member before it is composed into the program.

The second reason why this implementation lacks orthogonality is apparent from the output of two compositions. Both *PrintAdd* and *EvalVar* extend method *main* in the *Test* class so that, when composed in different order, these two modules execute the test operation in different order.

Both of these errors can be fixed easily. We leave the solution to a later section on Results of the Bali product line orthogonality test.

4.3 Orthogonality Test

The test for orthogonality can be expressed by a simplified 2-d matrix `A`. in Figure 4.3. Let sum (+) denote the composition operator. We need to prove that the program that is produced by summing rows and then columns is identical to the program that is produced by summing columns and then rows:

a_{11}	a_{12}
a_{21}	a_{22}

Figure 4.3 2D Cube *A*

$$a_{11} + a_{21} + a_{12} + a_{22} = a_{11} + a_{12} + a_{21} + a_{22} \quad (15)$$

By cancelling modules whose order is preserved, we can simplify (15) to be:

$$a_{21} + a_{12} = a_{12} + a_{21} \quad (16)$$

That is, the sum of the bottom left and upper right quadrants must commute. This is possible in FOP only if a_{21} and a_{12} satisfy the following conditions:

(a) they do not define or extend the same method or variable[1]

(b) they do not refer to members added by each other[2]

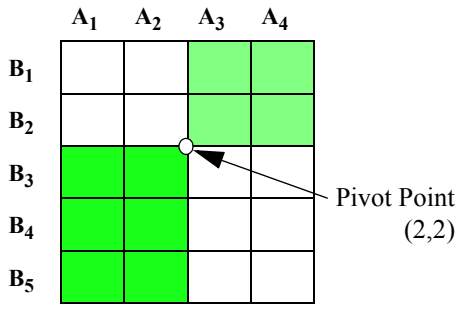


Figure 4.4 2D Model, H

4.3.1 Orthogonality of a Single Multidimensional Program

For any given program both conditions can be verified. The difficulty is doing so *efficiently* for larger multidimensional models. Consider model **H** of Figure 4.4. We can partition **H** into quadrants by choosing a *pivot point*s along each point in the grid. All we need to do is to verify that no element in the upper quadrant redefines, extends, or references a method or variable in any element of the lower quadrant, and vice versa. That is, there are no depencencies between two modules located at *right-diagonal*. We must perform this test for all elements in the shaded regions of **H**. Consider a 2-dimensional cube where both dimensions are of size *d*, then this is a $O(d^2)$ operation since there are $O(d^2)$ elements within the two quadrants. Furthermore, as there are $(d-1)^2$ pivot points in a $d*d$ matrix, verifying orthogonality using this method takes $O(d^4)$ time.

1.That is, each adds and extends non overlapping sets of members
2.Doing so would impose an ordering

For an *n*-dimensional cube, we must show that *each pair* of dimensions is orthogonal. To do this we have to choose every pair of dimensions (imagine that an *n*-dimensional cube has been collapsed to 2-dimensions) and apply the test in the previous paragraph. There are $\binom{n}{2} = O(n^2)$ dimension pairs, and each test of an *n*-dimensional model for all pivot points takes $O(d^4)$ time. So a brute force algorithm has an efficiency of $O(n^2d^4)$. For even reasonable values of *n* and *d*, brute-force is infeasible.

We devised a more efficient algorithm to test for orthogonality. Our insight: to test condition (a) requires two passes of the code-base. The first pass collects data on members that are defined in all features, and the second pass identifies member refinements. If cube entry $\mathbf{M}_{i,j..k}$ refines a member defined in feature $\mathbf{M}_{i',j'..k'}$, then we need to check if coordinates *(ij..k)* and *(i'j'..k')* fall into conflicting (shaded) quadrants for some pivot point. For *n* dimensions, there are $\binom{n}{2}$ dimension pairs to test, which is $O(n^2)$.

Verifying condition (b) can be accomplished in a similar way. The test is if cube entry $\mathbf{M}_{i,j..k}$ references a member defined in feature $\mathbf{M}_{i',j'..k'}$ then coordinates *(ij..k)* and *(i'j'..k')* cannot fall into conflicting (shaded) quadrants.

4.3.2 Orthogonality of a Product Line Model

Now consider a multidimensional (cube) model that represents an entire product line. Projections of this cube yield sub-cubes whose aggregation produces particular programs of the product line. As there can be an exponential number of possible sub-cubes, the problem of verifying the orthogonality of a cube is significantly harder.

We can test the orthogonality of a product-line cube by extending the algorithm of the previous section. The difference is an additional third condition. If cube entries $\mathbf{M}_{c1}$ and $\mathbf{M}_{c2}$ are found to conflict, where *c1* and *c2* are different *n*-dimensional coordinates, we need to know if $\mathbf{M}_{c1}$ and $\mathbf{M}_{c2}$ could ever be in the same product. (Cube entries *can* conflict if they *never* appear in the same program). We can make such a determination by using a SAT or CSP solver and the propositional formula representation of product-line's feature model.

Briefly, we know c_1 is an n-tuple of features $(\mathbf{f}_1, \mathbf{f}_2, \ldots \mathbf{f}_n)$, one feature per dimension. c_2 has a corresponding n-tuple $(\mathbf{g}_1, \mathbf{g}_2, \ldots \mathbf{g}_n)$. To see if these collections of features can appear in a product, we set their corresponding variables to be true in the propositional formula of our multidimensional model. If the SAT or CSP solver finds an assignment that satisfies the formula, we know $\mathbf{M}_{c1}$ and $\mathbf{M}_{c2}$ can appear together in a product.

Running a SAT or CSP solver can take time that is exponential in the number of variables. As feature models capture architectural abstractions of a product line, not only do they carry much fewer elements than its implementation, they change slowly compared to feature module source code. If a feature model hasn't changed, we can cache the results of a solver to remember particular feature compatibilities. In fact, multiple references between same feature module pairs need not be put through a SAT solver more than once. We simply cache results.

4.4 Results

Using the technique just described we have efficiently analyzed several AHEAD multidimensional product lines. TABLE 8. shows key statistics for Bali, Graph, and Java product-lines.

Product Line	Dim.	# of Features	# of Programs	Code Base Jak/Java LOC	Program Jak/Java LOC	Time to Run (seconds)
Bali Product Line	2	17	8	12K/16K	8K/12K	5
Graph Product Line	2	18	80	1800/1800	700/700	3
Java Product Line	3	70	56	34K/48K	22K/35K	20

TABLE 8. Product Line Statistics

Orthogonality validation was carried out on Pentium-4 3GHz with 1GB memory, running Windows XP Professional Edition. Using the technique described in the previous section we were able to verify a large product line with more than 30K LOC and 70 features within only 20 seconds. The brute force approach would have demanded much more

running time. The Java Matrix and GPL passed without any orthogonality failures. The Bali matrix on the other hand had several failures.

4.4.1 Bali

Bali is a grammar-specification language designed to produce parsers for AHEAD tools suite [10]. Unlike conventional techniques (lex/yacc), Bali supports composable grammar specification. Thus, a product line implemented in Bali represents a family of language parsers.

The implementation of Bali tools is itself designed to be a product line. The Bali model has two dimensions, one for language dialects and the other for tools. There are only two language dialects: *with*, and *without*, and 4 tools. Its AHEAD model is represented in Figure 4.5.

A matrix representation of Bali feature design is illustrated in Figure 4.6. Along the *language dialect* dimension, *Core* feature must be selected, leaving *withReqFeature* optional. Along the *tools* dimension the *Base* feature and one of the tools must be selected.

The first category of errors was due to a feature module referencing methods that are defined in another feature module located at right-diagonal. For instance, the module *require* was referencing *driver( )* method in a tester class defined within *bali2jak*, *bali2javacc*, *bali2layer*, and *composer*. As a consequence, language dialect dimension can not be aggregated before aggregating the tools dimension; *withReqFeature* depends upon presence of one of the tools. Fortunately there is an easy fix. Move the *driver()* method up

```
Bali : Lang Tools ;

Lang : core CoreFeatures ;
CoreFeatures : withRequireFeature
             | without;

Tools: base [codeGen] Tool :: BaliTools;
Tool : bali2jak :: Bali2JakTool
     | bali2javacc :: Bali2javaccTool
     | baliComposer :: BaliComposerTool
     | bali2layer :: Bali2layerTool;

%%
BaliComposerTool implies not codeGen;
Bali2JakTool or Bali2layerTool or Bali2javaccTool iff codeGen;
```

Figure 4.5 Bali Grammar

		TOOLS					
		Base	**CodGen**	**Bali2Jak**	**Bali2Javacc**	**Bali Composer**	**Bali2Layer**
LANG.	**Core**	kernel, bali, visitor, collect	codegen	bali2jak	bali2javacc	composer	bali2layer, b2lOptns, b2lGUI
	WithReqFeature	require		reqComposer	reqB2Javacc	reqComposer	
	Without						

Figure 4.6 Bali Matrix

to the *kernel* module, and let individual tools refine it. Hence, *require* module will reference a method defined in the *kernel* module.

We clarify this design fix in the figure below. Figure 4.7A contains features 1, 2, and 3 within a 2D matrix, where feature 3 references the *foo()* method that is defined in feature 2. When rows are composed before columns the resulting composition order is `Feature 1 + Feature 3 + Feature 2`. This leads to a back-reference, which is a violation of step-wise refinement paradigm: step-wise compositions should maintain consistent, working programs. We fix this in Figure 4.7B by introducing *foo()* method in Feature 1. Feature 3 can maintain its reference to *foo()* and the composition `Feature 1 + Feature 3` will compile.

The second category of errors arose from feature modules located at right-diagonal to each other refining the same method. As a consequence these two feature modules can not commute. Modules *require* and *codegen* were refining the same method, which is

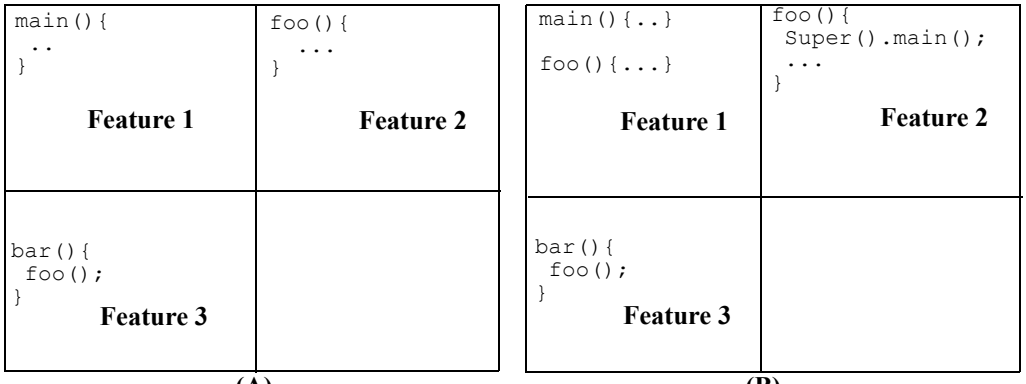


Figure 4.7 Fixing a Back References

defined in *bali*. In order to fix this problem we need to enforce order of operation and create non-overlapping members to be refined. An example follows.

Figure 4.8A shows implementation of three feature modules within a 2D matrix. Features 2 and 3 refine method *main()*, which is introduced in feature 1. Hence, features 2 and 3 do not commute. Aggregating along one dimension produces *a, b, c* order of operation and aggregating along the other produced *a, c, b* order.

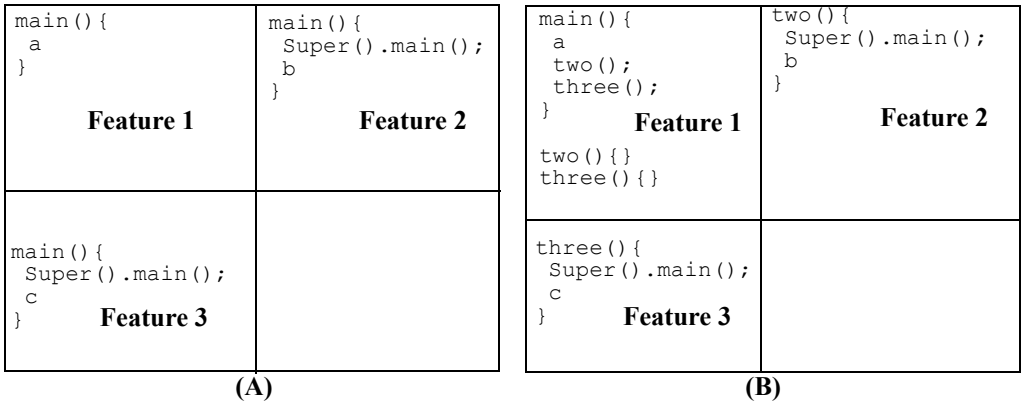


Figure 4.8 Fixing a Non-orthogonal Implementation

To fix this we introduce in feature 1 method *two()* to be refined by feature 2, and method *three()* to be refined by feature 3. Our fixed implementation is shown in Figure 4.8B. Now aggregating in either order executes operations in *a, b, c* order.

4.5 Perspective and Future Work

The two properties required for verifying orthogonality that we identified are an artifact of AHEAD's static-code composition technique. It is entirely dependent on mixin-based composition and the underlying implementation language semantics (i.e. java). For instance, verifying composition commutativity of AspectJ [68] modules may contain a different set of constraints. Likewise with other languages such as C# and C++. It is even more puzzling to imagine how one would verify commutativity of components that are composed at run-time, i.e., call each other through a well-defined interface. For example,

given three components `{A, B, C}` we want to verify that `A calls B calls C` is identical to `A calls C calls B`.

To verify commutativity of modules under different manner of composition we need to verify semantic equivalence of programs. One approach is to include predicates over some level of abstraction, and prove that both programs under consideration satisfy the predicates. Theorem provers such as Otter [58][5] have been used successfully to automate such proofs, however the task of verifying equivalence of programs is far from automated. Mulmuley in [59] develops theoretical grounds for semantic equivalence using an abstract algebraic model of typed lambda calculus. His model, Inclusive Predicate Logic, has an advantage of being mechanizable, yet as with other approaches, predicates are provided manually. A more promising approach to automate program verification is Symbolic Model Checking [60][61][62]. We believe that such techniques could be used to identify if within two programs transitions lead to different states, proving that they are semantically unequal.

We have circumvented proving semantic equivalence by identifying constraints that must hold in order for AHEAD composition to be equivalent. These constraints exploit the fact that syntactic commutativity implies semantic commutativity. Our constraints are simply an empirical insight; we can not prove that these two properties are sufficient to guarantee commutativity. To do so, we would need to analyze the semantics of operations within Java [20] or at least the more constrained Featherweight Java [26]. This proof is left as a direction for future work.

We have argued that under step-wise refinement a module can not reference or assume presence of any member of another module that has not been composed. This property provides us with clean composability semantics of function composition. For instance, consider variable *x* and functions *f, g* and *h* defined as follows:

x : *integer*

$f(y) : y^2$

$g(y) : y + 2$

$h(y) : cos(y)$

Assume a product line model permits only the following function compositions: `x`, `f(x)`, `g(f(x))`, and `h(g(f(x)))`. In the function composition `h(g(f(x)))`, *f* can not assume the presence of functions *g*, or *h*. So, *f* can not be defined in terms of *g* or *h* (e.g. *f(y)* : y^2 + *g(y)*). A function can, however, assume the presence of constructs from which it is built and that are not input parameters. Hence, *f* can be defined in terms of the square function and *h* can be defined in terms of *cos*.

When we introduce constraints to the above model, for instance: `f <=> g`, only three of the four compositions are valid: `x`, `g(f(x))`, and `h(g(f(x)))`. This constraint enforces functions *f* and *g* to be applied in a composed form. Let `j` be `f.g`, then the three valid compositions can be defined as `x`, `j(x))`, and `h(j(x))`. Functions *f* and *g* are not parameters to *j*, so *j* can assume the presence of *g* and *f*, showing that functions *g* and *f* can also assume each other's presence. For example, if *g(y)* is defined as *y+2*, and *f(y)* as y^2 + *g(y)*, composition `j(x)=g(f(x))` has the definition: $(x^2 + (x + 2)) + 2$. Function *f* back-references *g* but we are still able to resolve the complete definition of all function compositions in the product line. If *f* did not require *g*, we could not resolve *h(f(x))* because *g* is absent.

Put another way, back-references between two modules may be allowed when the referencing module requires the presence of the module being referred to.

With this insight we can relax the second requirement of orthogonality, that two modules located at right-diagonal of a matrix must not refer to members added by each other, if an only if, both modules are guaranteed by the model to be present in all products. In fact, we may safely remove the entire constraint since safe composition validation already reports any references to possibly undefined members.

4.6 Summary

As with other engineering disciplines Software Engineering aims to introduce greater predictability in software construction. Within the domain of product line designs, and in particular multidimensional designs, we have identified a key property that the soft-

ware should carry: *dimensional orthogonality.* The significance of this work is that we can automatically verify the orthogonality property of a multidimensional AHEAD design, thus demonstrating predictable results - that further aggregation of any dimensions, regardless of the order, will all result in a consistent view of the program.

The approach rests on proving commutativity of modules that are placed within sub-matrices at right-diagonals. We have showed how such verification can be carried out efficiently by identifying potential interferences between modules that must commute, and using SAT solvers to identify module compatibility. Furthermore, we have verified three AHEAD multidimensional product lines for orthogonality.

Given that multidimensional product lines have been built without verification of orthogonality, one may wonder what would have happened if a multidimensional model had not been orthogonal. In practice, we may find that with larger designs not all dimensional decompositions are completely orthogonal. This means one of two things: 1) our design can be improved to make the model orthogonal, or 2) we can restrict operations on the model.

While we have argued that orthogonality is an essential property, is not completely necessary for the software to work provided we forego certain operations. If a dimensional feature model's implementation lacks orthogonality property it simply means that we must impose an ordering among dimensions and we can not compose units of dimensions in arbitrary order. Hence, we no longer have the luxury of generating different views of the multidimensional model.

Furthermore, we argue that the utility of multidimensional models is limited without the ability to create multiple views of decompositions. We conclude that while it is certainly possible to create multidimensional models of product lines without ensuring the orthogonality property, in order to reap real benefits of this approach all multidimensional models must be validated for orthogonality.

Chapter 5

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.

AHEAD theory and tools are a realization of this idea. The contribution of this thesis has been to extend applicability of AHEAD's ideas by identifying and analyzing properties that product lines must maintain. First, we have identified a unique property between abstractions that warrants a higher-order structure in its design. Next, we described compositional properties that all product line implementations must carry. Doing so, we have shown that i) AHEAD's ideas scale to large code-bases, and ii) techniques for automating analysis of programs can be constructed with relative ease due to simplicity of AHEAD's compositional semantics.

This thesis answers the following questions:

- What properties must a design have to warrant a multidimensional structure?
- What are the properties necessary to ensure a product line lacks composition errors?
- How do we verify these properties?

We have argued that the answer to the first question revolves around orthogonal abstractions. Two abstractions are declared orthogonal if they equally influence each other. Put another way, further decomposition of either abstraction must be related to further decomposition of the other abstraction such that decomposing solely along either choice of abstraction yields an identical result. Orthogonal abstractions are synonymous with dimensions, where decomposition units of the abstraction forms the units along a dimension.

We have shown that even if a multidimensional design is thought to be orthogonal, its implementation may not be so. The idea behind verifying orthogonality is to show that decomposing along either of the abstraction produces the same program. Equivalently, we may also first decompose along both abstractions, and then verify whether aggregating one dimension is equivalent to aggregating the other. Under AHEAD's compositional model this verification amounts to test for commutativity of two feature module composition that lie at right-diagonal to each other.

We derive an algorithm for testing commutativity of feature modules that is more efficient than the naive brute-force approach. Results of orthogonality test on several existing multidimensional product lines revealed a flaw in Bali product line. It also confirmed our claim that our improved algorithm is tractable on larger product lines.

The other property that a product line must hold is lack of composition errors. Safe composition is the guarantee that a product line implementation is absent of composition errors. Within AHEAD, this property amounts to lack of references to undefined elements (classes, methods, variables) in a composed program's implementation for all programs in a product line. We have identified five constraints that must hold for any program being synthesized by the product line. Analysis of several existing product lines of respectable size revealed previously unknown errors and provided insight into creating better designs. Further, we were able to run these tests on 35K LOC product lines within 30 seconds, demonstrating feasibility of our approach on large code bases.

Even though our results are over programs of repectable size, the foundation of next-generation of software engineering practices will have to scale to orders-of-magni-

tude larger programs. We have argued that the principles of large-scale program designs will assume a clear mathematical form that simplifies their understanding and provides a disciplined way to think about their structure and properties.

Feature-Oriented Programming is one such structural theory in software engineering that aims to reduce complexity and reason at the level of feature constructs. We have demonstrated that by thinking at the structural level we can readily derive higher-order structural relationships, identify properties that a program's structure must hold, and reason about them in a much more efficient and automated manner. This work is but a step toward a science of software design and analysis.

Appendix A

Dimensions in Object Decomposition

6.1 Overview

We argued that multidimensional decomposition captures a structural relationship not seen easily within conventional hierarchal decomposition. Furthermore, we believe that these structural relationships surpass a particular manner of decomposition. Consequently, multidimensional relationships exist in Feature, Object, as well as Aspect decomposition. In this section we show benefits of discovering multidimensional structures within conventional Object-Oriented decompositions.

Decomposing a given problem into appropriate sub-problems relies at best on the designer's judgment and draws from past experiences. When a designer must decide between alternative choices, his knowledge of a decomposition's properties is typically used to guide his judgment. Multidimensional structures provide a clearer understanding of structural relationships within the domain, delineate design choices, and help clarify properties of a decomposition. We demonstrate this via two examples.

6.2 The Visitor Pattern

In Object-Oriented decomposition, a visitor pattern is a recurring problem of separating common operations from an object structure (class). The Visitor pattern promotes modifiability of operations - in particular adding new operations to existing object structures without modifying those structures.

The visitor pattern is a behavioral pattern. However, the solution itself has a structure. Its structure is commonly depicted by a UML class diagram. We demonstrate that these design patterns and their alternatives can be understood better given an appropriate multidimensional structure.

Recall the expression product line of Figure 2.7? Let the expression tree nodes be object structures, and let *print* and *evaluate* be operations we wish to factor out as visitors. Figure 6.1 shows the UML diagram of a solution using the Visitor design pattern.

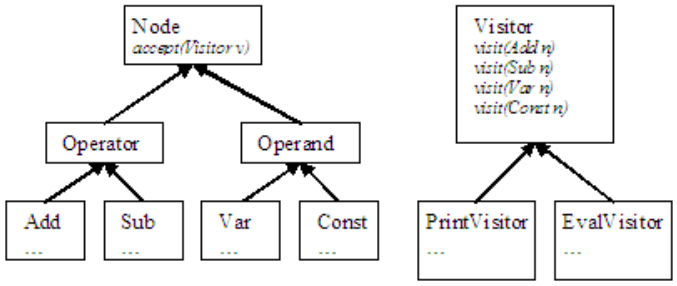


Figure 6.1 UML Diagram of a Visitor Pattern

Nodes *Add*, *Subtract*, *Variable*, and *Constant* form the structures that must be visited via an *accept* method. Given an instance of the expression `(x + 2) - 5`, to print it a *PrintVisitor* object is created and the root node *Sub* is visited by calling `Sub.accept(Visitor v)`. The *Sub* node calls visitor's Sub-specific visit operation, and passes the *Visitor* down to its children.

Abstractions *operations* and *node structures* in the expression evaluator are orthogonal. Operations *Print* and *Eval* must be performed on each node, and nodes *Add*, *Sub*, etc. require both operations *Print* and *Eval*. The matrix in Figure 6.2 captures this relationship. Note that *Node.** represents all other modules of the *node structure* dimension that are not listed explicitly. Likewise for *Visitor.**.

Node Structures

Operations	Node.*	Add	Sub	Var	Const
Visitor.*					
Print Visitor					
Eval Visitor					

Figure 6.2 Expression Evaluator Matrix

Given this matrix how must one structure the implementing classes? The visitor pattern's solution is shown in Figure 6.3, where each box represents a module (Class).

Nodes are fully decomposed into *Add*, *Sub*, *Var*, and *Const* classes. However, printing functionality is not fully decomposed. It is represented by the class *PrintVisitor*, which cohesively captures printing functionality, yet is not cohesive from the point of view of node structures because it contains functionality related to different node structures within the single class. The same holds with *Eval*. As a consequence, it promotes modifiability of operations. Adding or modifying an operation deals only with a single module. It is equivalent to adding a row in the matrix. On the other hand, adding a new node structure, such as *Multiply*, is not well supported. This is equivalent to adding a column in the matrix shown in Figure 6.3.

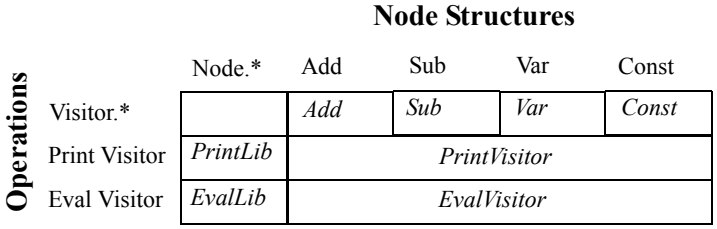


Figure 6.3 Expression Eval.'s Visitor Pattern Matrix

This trade-off in modifiability is apparent under the matrix view of orthogonal dimensions. Figure 6.4 shows an alternate decomposition. If modifiability of node structures it to be promoted, we would not decompose structure classes further into operations. For instance, the *Add* class will encapsulate all functionality related to the *Add* node.

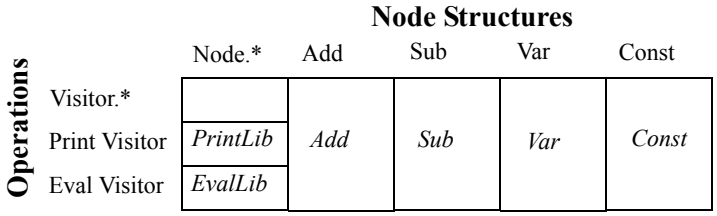


Figure 6.4 Expression Eval.'s Alternate Matrix

Note: AHEAD MDMs enable decomposing along both dimensions (Figure 6.2), and one would aggregate along the node structure dimension or the operations dimension to produce decomposition views similar to Figure 6.3 and Figure 6.4, respectively.

6.3 C++ Style Templates

Templates make certain classes and operation generic so that they can be applied on input of various types. Template markers such as `Queue<T>( )` can be read as Queue of type T, where the precise type of T is resolved at composition-time.

Consider an implementation of Containers *Queue* and *Stack*. *Queue* contains operations *enqueue*, *dequeue*, *clear*, etc. and *Stack* contains *push*, *pop*, *clear*, etc. Both must operate on an arbitrary set of objects *{A..Z}*. As with the Expression Evaluator, are dimensions *operations* and *type structures* orthogonal?

The Queue problem and the Expression Evaluator problem are similar in many respects but there are also some key differences. Both have a notion of structures and operations that seem to be orthogonal to each other. Within the Expression Evaluator the set of structures are finite and known, but typically a Queue operates on arbitrary object structure that may not be known apriori. Operations in Expression Evaluator are structure-specific (print an *Add* node, print a *Var* node), whereas Queuing operations are not (enqueue an object by adding a pointer to it). Hence, the code for enqueuing, dequeuing, etc. will not change as new structures are added. It is for this reason that the second dimension must not be operations, but containers (i.e., one abstraction up). Operation of a Queue are not influenced by different object types! Containers, on the other hand, are orthogonal to Object Types. Object types *{A..Z}* require *Queue* and *Stack* containers, and each Container must contain Object type-specific data - Queue of type A, Stack of type B, etc.

Hence, the visitor pattern is not adequate, it will replicate code for operations. Templates solve this problem easily by making the container classes and their operations generic. One way of understanding this solution is by considering its orthogonal dimensions. Figure 6.5 shows a matrix where Containers and Object Type are orthogonal. Black boxes show modules that are implemented by the programmer. Greyed boxes show modules generated by the STL preprocessor.

We have shown that one way to understand solutions that templates provide is by considering the multidimensional structure of the problem domain. Templates essentially

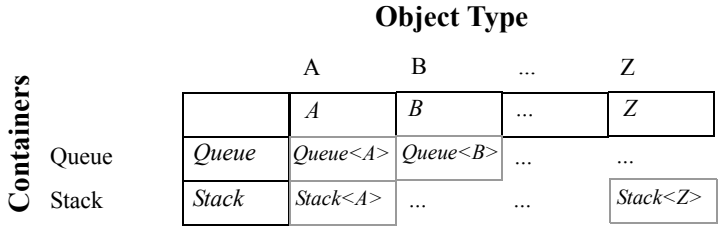


Figure 6.5 Solution Using Templates

allow an object-oriented decomposition to factor out a functionality common among multiple modules - here Queuing and Stacking. Extending this view, templates of templates - for instance, `Queue<Stack<T>>`, are 3-dimensional orthogonal relationships.

Template specialization also fits in with this view. Specialization simple corresponds to manually providing code for a module that is generated by the preprocessor generator - i.e. one of the greyed boxes in Figure 6.5.

6.4 Perspective

Orthogonal dimensions are abundant in nature. Here we have surveyed the Visitor design pattern and Templates. We have also discovered similar orthogonal dimensions in several other design patterns, for instance: Abstract Factory, and Model-View-Controller.

Many of the tools and techniques that have been developed to counter short-comings of object-oriented decomposition can be understood in terms of multidimensional structures. Tradeoffs in modifiability are typically - but not always - symptomatic of multidimensional structural relationship between abstractions. Discovering such dimensional relationships within a problem's decomposition structure helps understand the decomposition, assists in deciding which technique should be used to solve a problem, and how to structure the modules that implement orthogonal abstractions.

We have touched only the surface of these ideas. We believe there are many more examples to be studied and lessons to be learnt for techniques of multidimensional abstractions to be mastered.

Bibliography

[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. “Verification of Feature-Based Model Templates Against Well-Formedness OCL Constraints”. Technical Report 2005-19, Department of Electrical and Computer Engineering, University of Waterloo, 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/java-world/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.

[53] P. A. Tuscany. “Software development environment for large switching projects.” In *Proceedings of Software Engineering for Telecommunications Switching Systems Conference*, 1987.

[54] D. E. Perry. Dimensions of Consistency in Source Versions and System Compositions, In *Proc. of the 3rd Intl. Workshop on Software Configuration Management*, pages 29-32, 1991.

[55] Perry, D. E.: System Compositions and Shared Dependencies. *Proceedings of SCM-6Workshop*, ICSE’96, Berlin, Germany. LNCS, Springer-Verlag 1996.

[56] D. E. Perry. A Product Line Architecture for a Network Product. In *Proceedings of the Third International Workshop on Software Architecture for Product Families*, pages 41–54, Mar. 2000.

[57] J. Gray et al., “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals,” *Proc. 12th Int’l Conf. on Data*

Engineering, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 152-159.

[58] Kalman, John Arnold. *Automated Reasoning with OTTER*.

[59] Mulmuley, Ketan. Full Abstraction and Semantic Equivalence, MIT Press, Feb 1987.

[60] Burch, J, et al., "Symbolic model checking: 10^{20} states and beyond", Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, 1990.

[61] Holzmann, G.J. "The Model Checker SPIN", IEEE Transactions on Software Engineering, 1997.

[62] By Edmund M. Jr. Clarke, Orna Grumberg, Doron A. Peled, "Model Checking", MIT Press, 2000.

[63] Apache Ant Project, http://ant.apache.org

[64] 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.

[65] Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley (2004)

[66] Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE (1999) 107-119

[67] Batory, D., Liu, J., Sarvela, J.N.: Refinements and Multidimensional Separation of Concerns. ACM SIGSOFT, September (2003)

[68] AspectJ Version 1.2. http://eclipse.org/aspectj

Vita

Sahil Thaker was born in Mumbai, India on November 6, 1981, the son of Prakash Thaker and Bharati Thaker. In 1996 he migrated with his family to Auckland, New Zealand, where he attended remainder of his high-school studies. While at high-school, Sahil co-founded a garage-company, Virtualforce, providing web hosting and programming services to local clients.

Following high-school, Sahil interned at AMS, a small business software company, as a web developer, and subsequently joined the Engineering program at University of Auckland with the Portage Licensing Trust Scholarship. During undergraduate years his interests magnified towards software engineering. He sold Virtualforce to other partners and founded Webdevelopers Ltd. Under the hood of Webdevelopers, he carried out numerous e-commerce and workflow system projects. At one point the company grew two-fold from one person to two.

Sahil graduated from the University of Auckland at the end of 2003 with Bachelor of Engineering in Software Engineering with First Class Honors, and joined a local software development firm, ECONZ, as a Software Engineer. Mystified by the science in Software Engineering, he decided to pursue graduate studies in Computer Science. Sahil joined the graduate school at University of Texas-Austin in Spring 2005 where he pursued his Masters degree.

Permanent Address: 10 Walworth Avenue,
Pakuranga, Auckland, New Zealand

This thesis was typed by the author.