Selected Papers

- Achieving Extensibility through Product-Lines and Domain-Specific Languages, *ACM TOSEM 200*2. Topic: Describes the FSATS project and its use of domain-specific languages and refinements.

- Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs, *ACM TOSEM 2002*. Topic: Explains the mixin-layer technique for implementing feature refinements.

- Design Wizards and Visual Programming Environments for GenVoca Generators, *IEEE TSE May 2000*. Topic: automatic optimization of GenVoca equations, also known as automatic programming.

- A Standard Problem for Evaluating Product-Line Methodologies, *GCSE 2001*. Topic: a good introduction to GenVoca and Feature-Oriented Programming.

- JTS: Tools for Implementing Domain-Specific Languages, *ICSR June 1988*. Topic: tools for implementing GenVoca models and preprocessors for extensible Java languages.

- Object-Oriented Frameworks and Product-Lines. *SPLC August 1999*. Topic: how customized OO frameworks can be synthesized using feature refinements.

- Generating Product-Lines of Product-Families, *Automated Software Engineering 2002*. Topic: Origami and scaling feature refinements to multiple programs.

- Scaling Step-Wise Refinement, Submitted. Topic: The AHEAD model and how to generalize step-wise refinement to synthesize consistent code and non-code artifacts.

Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study

DON BATORY, CLAY JOHNSON, BOB MACDONALD, and DALE VON HEEDER
University of Texas at Austin

This is a case study in the use of *product-line architectures* (*PLAs*) and *domain-specific languages* (*DSLs*) to design an extensible command-and-control simulator for Army fire support. The reusable components of our PLA are layers or "aspects" whose addition or removal simultaneously impacts the source code of multiple objects in multiple, distributed programs. The complexity of our component specifications is substantially reduced by using a DSL for defining and refining state machines, abstractions that are fundamental to simulators. We present preliminary results that show how our PLA and DSL synergistically produce a more flexible way of implementing state-machine-based simulators than is possible with a pure Java implementation.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies* (e.g., object-oriented, structured); D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Evolutionary prototyping*; *State diagrams*; D.2.10 [**Software Engineering**]: Design—*Methodologies and representations*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specfic architectures*; *Languages*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*; D.2.m [**Software Engineering**]: Miscellaneous—*Rapid prototyping*

General Term: Design

Additional Key Words and Phrases: GenVoca, domain-specific languages, simulation, aspects, refinements

1. INTRODUCTION

Software evolution is a costly yet unavoidable consequence of a successful application. Evolution occurs when new features are added and existing capabilities

This work was supported in part by the University of Texas Applied Research Labs and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

Preliminary version presented at the International Conference on Software Reuse, Vienna, Austria, July 2000. Updated for ACM TOSEM September 2001.

Authors' addresses: D. Batory, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712; email: batory@cs.utexas.edu; C. Johnson, B. MacDonald, and D. von Herder, Applied Research Labs, University of Texas at Austin, Austin Texas 78713; email: {clay, bob, drv}@arlut.utexas.edu.

are enhanced. Unfortunately, many applications suffer *design fatigue*—when further evolution is difficult and costly because of issues not addressed in the initial design [Graves 2001]. Creating software that is easily evolvable is a central problem today in software engineering.

Three of several proposed complementary technologies address software evolution: object-oriented design patterns, domain-specific languages, and product-line architectures. *Design patterns* are techniques for restructuring and generalizing object-oriented software [Gamma et al. 1995]. Evolution occurs by applying design patterns to an existing design; the effects of these changes are borne by programmers who must manually transform an existing code base to match the updated design. Recent advances indicate that tool support for automating the applications of patterns is possible [Tokuda and Batory 1999]. *Domain-specific languages* (*DSLs*) raise the level of programming to allow customized applications to be specified compactly in terms of domain concepts; compilers translate DSL specifications into source code. Evolution is achieved by modifying DSL specifications [Van Deursen and Klint 1997]. *Product-line architectures* (*PLAs*) are designs for families of related applications; application construction is accomplished by composing reusable components. Evolution occurs by plugging and unplugging components that encapsulate new and enhanced features [Batory 1998; Bosch 1999; Czarnecki and Eisenecker 1999; Software Engineering Institute 2001; Weiss and Lai 1999]. Among PLA models, the GenVoca model is distinguished by an integration of ideas from aspect-oriented programming [Kiczales et al. 1997], parameterized programming [Goguen 1986], and program-construction by refinement [Baxter 1992].

This paper presents a case study in the use of GenVoca PLAs and DSLs to create an extensible command-and-control simulator for Army fire support. (Design patterns were also used, but they played a minor role.) We discovered that components of distributed simulations are *not* conventional DCOM and CORBA components, but rather are layers or “aspects” whose addition or removal simultaneously impacts the source code of *multiple*, distributed programs. Further, we found that writing our components in a general-purpose programming language (Java) resulted in complex code that obscured a relatively simple, state-machine-based design. By extending Java with domain-specific abstractions (in our case, state-machines), our component specifications were more readily understood by domain experts, knowledge engineers, and application programmers. Thus, this case study is interesting not only because of the novelties introduced by PLAs and DSLs, but also because of their integration: using only one technology would have been inadequate.

We begin by explaining the ideas and terminology of fire support. We review an existing simulator, called FSATS, and motivate its redesign. We present a GenVoca PLA for creating extensible fire-support simulators and introduce an extension to the Java language for defining and refining state-machines. Finally, based on simple measures of program complexity, we show how PLAs and DSLs individually simplify simulators, but only their combination provides practical extensibility.

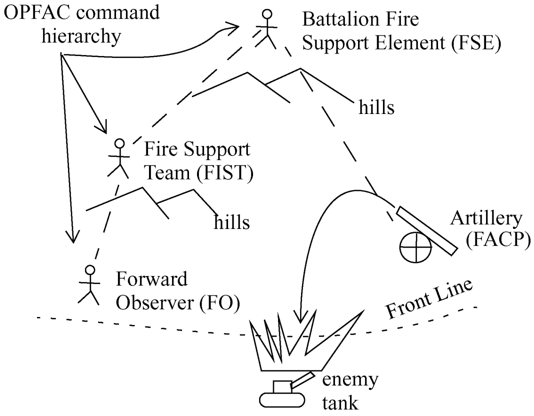


Fig. 1. OPFAC command hierarchy.

2. BACKGROUND

2.1 The Domain of Fire Support

Fire support is a command-and-control application that includes the detection of targets, assignment of weapons to attack the target, and coordination of the actual attack. The entities engaged in this process, called *operational facilities* (*OPFACs*), are soldier-operated (not machine-automated) command posts that exchange tactical (theater-of-war) messages.

Forward observers (*FO*) are OPFACs that are stationed at intervals across the frontline of a battlefield (Figure 1). They are one of several kinds of sensors responsible for detecting potential targets. A hierarchy of *fire support elements* (*FSE*) is responsible for directing requests from FOs to the most appropriate weapon system to handle the attack. FOs report to their *fire support team* (*FIST*); a FIST reports to a battalion FSE, a battalion FSE reports to a brigade FSE, and so on. Each FSE typically has one or more supporting *command posts* (*CPs*) with different weapon systems. For example, a battalion FSE might be supported by a *field artillery command post* (*FACP*); a FIST might be supported a mortar command post, and so on. In general, higher echelon FSEs are supported by higher echelon CPs with more powerful and/or longer range weapon systems.

FOs, FISTs, and other FSEs are responsible for evaluating a target. An evaluation may result in (a) assigning the target to be attacked by a supporting weapon, (b) elevating the target to the next higher echelon FSE for evaluation, or (c) denial—choosing not to attack the target. CPs are responsible for assigning targets to the best weapon or combination of weapons under their command. Once weapon(s) are assigned, messages are exchanged with the mission originator (usually an FO) to coordinate the completion of the mission. The particular message sequence depends on the target and weapon. It is still generally the case that all messages are relayed along the chain of CPs and FSEs that were involved in initiating the mission, although newer systems permit messages to be exchanged directly between the weapon and observer. The message sequence for a particular mission is referred to as the *mission thread*. In general, an OPFAC can participate in any number of mission threads at one time.

A mission thread is an instance of a *mission type*. There are well over twenty mission types, including:

- *when-ready-fire-for-effect-mortars* (*WRFFE-mortars*)—a mortar CP is assigned to shoot at a target as soon as possible,
- *when-ready-fire-for-effect-artillery* (*WRFFE-artillery*)—one or more artillery CPs are assigned to shoot at a target as soon as possible,
- *time-on-target-artillery* (*TOT-artillery*)—field artillery are requested to fire at a target so that all rounds land at the specified location at the specified time, and
- *when-ready-adjust-mortars* (*WRAdjust-mortars*)—a forward observer knows only approximately the location of the enemy and requests single rounds to be fired with the observer sending corrections between rounds until the target is hit, at which point it becomes a WRFFE-mortar mission.

Each OPFAC (FO, FIST, FSE, etc.) performs different actions for each mission type. For example, the actions taken by an FO for a TOT-mortar mission are different than those for a WRFFE-artillery mission.

Clearly, the above description of fire support is highly simplified, for example, the actions taken by specific OPFACs in a mission thread and the translation of messages into formats for tactical transmission were omitted. These details, however, are not needed to understand the contributions of this paper.

2.1.1 *FSATS.* Simulation plays a key role in U.S. Army testing and training. It avoids costs of mobilizing live forces, provides repeatability in testing, and allows force-on-force combat training without the liability. Simulation has been used to model virtual environments, weapons effects, outcome adjudication, and as computational resources increase, the fidelity has been refined to entity-level simulators.

Fire support is one of a number of domains that has been modernized by digital *Command*, *Control*, *Communications*, *Computer*, *and Intelligence* (*C4I*) systems that automate battlefield mission processing. *AFATDS* (*Advanced Field Artillery Tactical Data System*) is arguably the most sophisticated C4I system in existence, and provides the software backbone (message transmission, processing, etc.) for fire support for the Army [Magnavox 1999]. *FSATS* (*Fire Support Automated Test System*) is a system for testing AFATDS and other fire-support C4I systems. FSATS collects digital message traffic from command and control communication networks, interprets these messages, and stores them in a database for later analysis. FSATS can simulate any or all OPFACs used in AFATDS [FSATS 1999]. The subject of a test can be overall system performance, individual OPFAC performance, or system operator performance. Thus, FSATS is used both in training Army personnel in fire support and debugging/testing AFATDS.

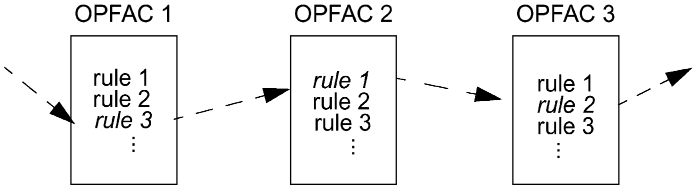


Fig. 2. Rule sets vs. mission threads.

2.1.2 *The Current FSATS Implementation.* FSATS has been under development for almost ten years. It is typical of the systems mentioned in our introductory paragraph: it began with a clean design but as its capabilities were extended, limitations of that design became increasingly troublesome.

The implementation is a combination of decision rules encoded in database tables, a set of "common actions" written as Ada procedures, and a decision rule interpreter, also in Ada. One set of rules is associated with each pair of an OPFAC type and a message type. When a tactical message is received by an OPFAC, the appropriate rule set is selected by the interpreter and each rule in the set is sequentially evaluated until one succeeds, at which point the action for that rule is executed and processing of that message terminates. There are from 200 to 1000 rules associated with each OPFAC type, divided among the various input message types. Each rule consists of a predicate, which is a conjunction of guards, and an action which is an index to a sequence of state and message common actions. Predicates typically contain five to ten guards (terms). The processing of rule sets is optimized, so that predicates can assume the failure of all previous predicates. Common actions range from simple (copy the target number field from the input to the output message) to complex (determining whether there exists a supporting OPFAC of type mortar which is capable of shooting the target indicated by the current message).

There are now obvious drawbacks to this design/implementation. While rule sets are used to express OPFAC behavior, OPFAC behavior is routinely understood and analyzed in terms of mission threads. Figure 2 illustrates a mission thread, the horizontal execution path, that associates various rules spanning multiple OPFAC programs. This complicates the knowledge acquisition and engineering process to derive from an analysis of multiple mission threads the rules as they apply at each OPFAC. Conversely, it obfuscates analyzing and debugging system behavior where rules for multiple mission threads are merged into monolithic sets within each OPFAC program.

The contrast of the vertical nature of rule sets versus the horizontal or "crosscutting" nature of mission threads in Figure 2 illustrates an encapsulation dichotomy that is not unique to FSATS [Batory and O'Malley 1992; Kiczales et al. 1997; Reenskaug et al. 1992]. In general, conventional OO approaches explore *use cases* (threads) for specification and analysis of system behavior. However, the concept of a use case is transient in a design process that identifies behavior (rules) with the actors (OPFACs) rather than the actions (missions). This trade-off is seemingly unavoidable given the need to produce objects that combine behaviors to react to a variety of situations. In FSATS, the transformation of mission threads into rule sets yields autonomous OPFACs at an increased cost to analysis and maintenance.

As FSATS evolved, rule sets quickly became large and unwieldy. Moreover, different missions might use the same message type at an OPFAC for slightly different purposes. Simpler rules that once sufficed often had to be factored to disambiguate their applicability to newer, more specialized missions. In worse cases, large subsets of rules had to be duplicated, resulting in a dramatic increase in rules and interactions. Moreover, the relationship between rules of different OPFACs, and the missions to which they applied, was lost. Modifying OPFAC rules became perilous without laborious analysis to rediscover and reassess those dependencies. The combinatorial effect of rule set interactions made analysis increasingly difficult and time-consuming.

FSATS management realized that the current implementation was not sustainable in the long term, and a new approach was sought. FSATS would continue to evolve through the addition of new mission types and by varying the behavior of an OPFAC or mission to accommodate doctrinal differences over time or between different branches of the military. Thus, the clear need for extensible simulators was envisioned. The primary goals of a redesign were to:

- disentangle the logic implementing different mission types to make implementation and testing of a mission independent of existing missions,
- reduce the "conceptual distance" from logic specification to its implementation so that implementations are easily traced back to requirements and verified, and
- allow convenient switching of mission implementations to accommodate requirements from different users and to experiment with new approaches.

2.2 GenVoca

The technology chosen to address problems identified in the first-generation FSATS simulator was a GenVoca PLA implemented using the *Jakarta Tool Suite* (*JTS*) [Batory et al. 1998]. In this section, we motivate and explain basic ideas of GenVoca and one of its implementation techniques. It is beyond the scope of this paper to review design methodologies (i.e., how to apply GenVoca concepts) or to explain domains simpler than FSATS to elaborate the approach that we have taken. Interested readers should consult [Smaragdakis and Batory 2002], [Batory et al. 1995], and [Lopez-Herrejon and Batory 2001].

2.2.1 *Motivation.* Today's models of software are too low-level, exposing classes, methods, and objects as the focal point of discourse in software design and implementation. This makes it difficult, if not impossible, to reason about software architectures (a.k.a. component-based designs), to have simple, elegant, and easy to understand specifications of applications, and to be able to create and critique software designs automatically, given a set of high-level requirements.

Simple specifications that are amenable to automated reasoning, code generation, and analysis, are indeed possible provided that the focus of discourse can be shifted to components that encapsulate the implementation of individual and largely orthogonal *features* that can be shared by multiple applications.[1] The intuitive rationale for this shift is evident in discussions about software products: architects don't speak about their products in terms of code modules, but instead explain their products in terms of features offered to clients. That is, the focus of discourse is on features and not on source code. GenVoca aims to raise the level of abstraction of understanding software from code modules (or code-encapsulation technologies) to features (or feature-encapsulation technologies).

2.2.2 *Features and Refinements.* At its core, GenVoca is a design methodology for creating product-lines and building architecturally-extensible software—software that is extensible via component additions and removals. GenVoca is a scalable outgrowth of an old and practitioner-ignored methodology called *step-wise refinement*, which advocates that efficient programs can be created by revealing implementation details in a progressive manner. Traditional work on step-wise refinement focussed on microscopic program refinements (e.g., $\texttt{x} + \texttt{0} \Rightarrow \texttt{x}$), for which one had to apply hundreds or thousands of refinements to yield admittedly small programs. While the approach is fundamental, and industrial infrastructures are on the horizon [Baxter 1992; Simonyi 1995], GenVoca extends step-wise refinement by scaling refinements to a component or layer (i.e., multi-class-modularization) granularity, so that each refinement adds a feature to a program, and composing a few refinements yields an entire application.

The critical shift to understand software in this manner is to recognize that programs are *values*, and that refinements are *functions* that add features to programs. Consider the following constants (parameterless functions) that represent programs with different features:

```
f()    //program with feature f
g()    //program with feature g
```

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

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

It follows that a multi-featured application is specified by an *equation* that is a named composition of functions, and that different equations define a family of applications, such as:

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

Thus, by casually inspecting an equation, one can readily determine features of an application.

[1] Griss [2000] defines a *feature* as a product characteristic that users and customers view as important in describing and distinguishing members of a product-line.

Note that there is a subtle but important confluence of ideas in this model: a function represents both a feature *and* its implementation. Thus, there can be different functions that offer different implementations of the *same* feature:

$\mathtt{k}_1\mathtt{(x)}$ //adds feature k (with $\mathtt{implementation}_1$) to x

$\mathtt{k}_2\mathtt{(x)}$ //adds feature k (with $\mathtt{implementation}_2$) to x

So when an application requires the use of feature k, it becomes a problem of *equation optimization* to determine which implementation of k would be the best (e.g., provide the best performance).[2] It is possible to automatically design software (i.e., produce an equation that optimizes some qualitative criteria) given a set of declarative constraints for a target application. An example of this kind of automated reasoning is presented in Batory et al. [2000].

As a practical matter, refinements typically cannot transform arbitrary programs. Rather, the input to refinements (functions) must satisfy a *type*—a set of constraints that are both syntactic and semantic in nature. A typical syntactic constraint is that a program must implement a set of well-defined Java interfaces; a typical semantic constraint is that the implementation of these interfaces satisfy certain behavioral properties. Thus, it is common that not all combinations of features (or their implementations) are correct [Kang et al. 1990]. A model for expressing program types and algorithms that can automatically and efficiently validate equations has been developed and is part of the Jakarta Tool Suite [Batory and Geraci 1997].

2.2.3 *Mixin-Layer Implementation.* There are many ways in which to implement refinements, ranging from dynamically composing objects to statically-composed meta-programs (i.e., programs that generate other programs) [Batory et al. 1998] and rule-sets of program transformation systems [Neighbors 1997]. One of the simplest is to use templates called *mixin-layers*. In the following, we use the term *component* to denote a mixin-layer implementation of a refinement.

A GenVoca component typically encapsulates multiple classes. Figure 3a depicts component X with four classes A–D. Any number of relationships can exist among these classes; Figure 3a shows only inheritance relationships. That is, B and C are subclasses of A, while D has no inheritance relationship with A–C.

The concept of refinement is an integral part of object-orientation. In particular, a subclass is a *refinement* of its superclass: it adds new data members, methods, and/or overrides existing methods. A *GenVoca refinement* scales inheritance to simultaneously refine multiple classes.[3] Figure 3b depicts a component Y that encapsulates three refining classes (A, B, and D) and an additional class (E). Note that the refining classes (A, B, D) do not have their superclasses

[2] Technically, different equations represent different programs. Equation optimization is over the space of semantically equivalent programs. This is identical to relational query optimization: a query is initially represented by a relational algebra expression, and this expression is optimized. Each expression represents a different, but semantically equivalent, query-evaluation program as the original expression.

[3] There are other kinds of refinements beyond those discussed in this paper. An example is an optimizing refinement, which maps an inefficient program to an efficient program [Neighbors 1997].

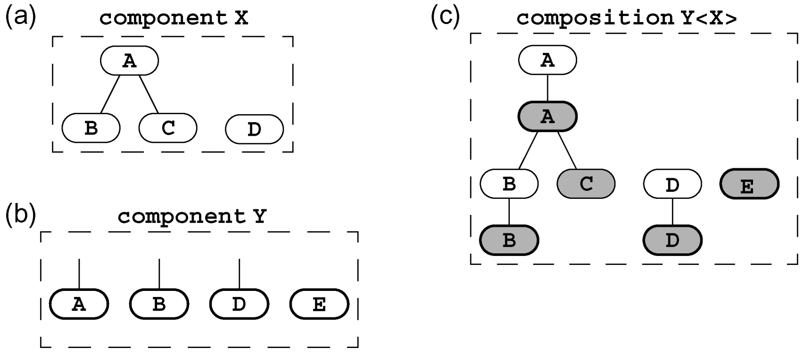


Fig. 3. GenVoca components and their composition.

specified; this enables them to be "plugged" underneath their yet-to-be-determined superclasses.[4]

In our model, where refinements are functions, we would write the composition of Y with X as Y(X). When dealing with template implementations, however, the convention is to use a slightly different syntax, Y<X>. Thus, there is a trivial correspondence between model equations and their implementing mixin-layer template expressions.

Given this correspondence, Figure 3c shows the result of Y<X>. (The classes of Y are outlined in darker ovals to distinguish them from classes of X). Note that the obvious thing happens to classes A, B, and D of component X—they are refined by classes in Y, as expected. That is, a linear inheritance *refinement chain* is created, with the original definition (from X) at the top of the chain, and the most recent refinement (from Y) at the bottom. As more components are composed, the inheritance hierarchies that are produced get progressively broader (as new classes are added) and deeper (as existing classes are refined). As a rule, only the bottom-most class of a refinement chain is instantiated and subclassed to form other distinct chains. (These are indicated by the shaded classes of Figure 3c). The reason is that these classes contain all of the "features" or "aspects" that were added by higher classes in the chain. These higher classes simply represent intermediate derivations of the bottom class [Batory et al. 1998; Findler and Flatt 1998; Smaragdakis and Batory 1998]. A consequence of instantiating the "bottom-most" class of a chain is that refinement relationships take precedence over typical subclassing relationships. That is, if class A in component X is refined, it is the *most refined* version of A that is the superclass of B. This precedence relationship can be seen in Figure 3c.

Representation. A GenVoca component/refinement is encoded in JTS as a class with nested classes. A representation of component X of Figure 3a is shown below, where $TEqn.A denotes the most refined version of class A (e.g.,

[4] More accurately, a *refinement* of class A is a subclass of A with name A. Normally, subclasses must have names distinct from their superclass, but not so here. The idea is to graft on as many refinements to a class as necessary—forming a linear "refinement" chain—to synthesize the actual version of A that is to be used. Subclasses with names distinct from their superclass define entirely new classes (such as B and C above), which can subsequently be refined.

classes `X.B` and `X.C` in Figure 3a have `$TEqn.A` as their superclass). We use the Java technique of defining properties via empty interfaces; interface `F` is used to indicate the type of component `X`:

```
interface F { }    // empty

class X implements F {
  class A {...}
  class B extends $TEqn.A {...}
  class C extends $TEqn.A {...}
  class D {...}
}
```

Components like `Y` that encapsulate refinements are expressed as mixins—classes whose superclass is specified via a parameter. A representation of `Y` is a *mixin-layer* [Findler and Flatt 1998; Smaragdakis and Batory 1998, 2002], where `Y`'s parameter `s` can be instantiated by any component that is of type `F`:

```
class Y < F s > extends s implements F {
  class A extends s.A {...}
  class B extends s.B {...}
  class D extends s.D {...}
  class E {...}
}
```

In the parlance of the model of Section 2.2.2, `X` is a value of type `F`, and `Y` is a function with a parameter `s` of type `F` that returns a refined program of type `F`. The composition of `Y` with `X`, depicted in Figure 3c, is expressed by:

```
class MyExample extends Y<X>;
```

where `$TEqn` is replaced by `MyExample` in the instantiated bodies of `X` and `Y`. Readers familiar with earlier descriptions of the GenVoca model will recognize that `F` corresponds to a realm interface,[5] `X` and `Y` are components of realm `F`, and `MyExample` is a type equation [Batory and O'Malley 1992]. Extensibility is achieved by adding and removing mixin-layers from applications; product-line applications are defined by different compositions of mixin-layers.

2.2.4 *Perspective.* Stepwise refinement originated in the late-1960 writings of Wirth and Dijkstra. The key to its modernization lies in scaling the effects of individual refinements, to which there are many contributors. Neighbors [1989] first described the architectural organization of mapping from abstract to concrete languages in DRACO, where the mappings between

[5] Technically, a realm interface would not be empty, but would specify class interfaces and their methods. That is, a realm interface would include nested interfaces of the classes that a component of that realm should implement. Thus, nested class `A` of `Y` would extend `s.A` as above, but also might implement `F.IA`, a particular nested interface of `F`. Java (and current JTS extensions of Java) do not enforce that class interfaces be implemented when interface declarations are nested [Smaragdakis and Batory 1998]. On going research aims to correct this situation [Cardone and Lin 2001].

a higher (more abstract) language representation of a program to a lower (more implementation-oriented) representation can be seen as large-scale refinements. Parameterized programming, which provides the conceptual infrastructure for early models on parametric components, was advanced by Goguen [1986]. The earliest use of plug-compatible layers (i.e., large-scale refinements) for creating product families and extensible applications originated in the mid-to-late 1980s in the work of Batory and O'Malley [1992]. Feature descriptions of applications and product-lines originated in the early 1990s with Kang's [1990] FODA (Feature Oriented Domain Analysis) and Gomaa's EDLC (Evolutionary Domain Life Cycle) [Gomaa et al. 1992] models. Collaborations, as object-oriented representations of refinements, were discussed by Reenskaug in 1992 [Reenskaug et al. 1992]. Kiczales's notion of aspects with "cross-cutting" effects clarified the general need for feature encapsulations [Kiczales et al. 1997]. Recent work on multi-dimensional separation of concerns examines a more flexible way of identifying and composing features in existing software [Tarr et al. 1999].

It is also worth noting the trade-off between the large-scale refinements of GenVoca and generic small-scale (or microscopic) refinements ($x + 0 \Rightarrow x$) that are more commonly found in the literature (e.g., [Rich and Waters 1992]).

The traditional argument for small-scale refinements is that a relatively small number of generic small-scale refinements can generate a larger number of large-scale refinements. Additionally, large-scale refinements tend to be applicable less often, because they tend to make more assumptions about the application context. (That is, the refinement Y of the Figure 3 is applicable less often than a "sub-refinement" that only specializes A, because Y requires the presence of B and C.) Where the case for traditional small-scale refinements breaks down is precisely when doing domain-specific development; the generation argument fails because hardly any of the generic transforms are of interest in a restricted domain, and the contextual assumptions argument breaks down because the domain provides the required context.

Domain-specific small-scale refinements can indeed be used to address the above-cited deficiencies. But, as we mentioned earlier, enormous numbers of domain-specific small-scale refinements must be applied to produce admittedly small programs. Scaling refinements, as we are doing, provides a more practical way to develop complex, domain-specific software artifacts. The tools are simpler, and the concepts are closer to main-stream programming methodologies (e.g., OO collaborations, as we will see in the next section).

3. THE IMPLEMENTATION

The GenVoca-FSATS design was implemented using the *Jakarta Tool Suite* (*JTS*) [Batory et al. 1998], a set of Java-based tools for creating product-line architectures and compilers for extensible Java languages. The following sections outline the essential concepts of our JTS implementation.

3.1 A Design for an Extensible Fire-Support Simulator

The Design. The key idea behind the GenVoca-FSATS design is the encapsulation of individual mission types as components. That is, the central

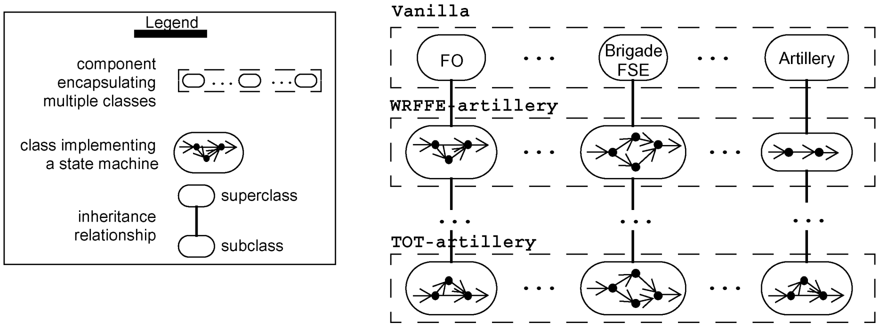


Fig. 4. OPFAC inheritance refinement hierarchy.

variabilities in FSATS throughout its history (and projected future) lie in the addition, enhancement, and removal of mission types. By encapsulating mission types as components, evolution of FSATS is greatly simplified.

We noted earlier, that every mission type has a “cross-cutting effect”, because the addition or removal of a mission type impacts multiple OPFAC programs. A mission type is an example of a common kind of refinement called a *collaboration*—a set of objects that work collectively to achieve a certain goal [Reenskaug et al. 1992; Smaragdakis and Batory 1998; Van Hilst and Notkin 1996]. Collaborations have the desirable property that they can be defined largely in isolation from other collaborations, thereby simplifying application design. In the case of FSATS, a mission is a collaboration of objects (OPFACs) that work cooperatively to prosecute a particular mission. The actions taken by each OPFAC are defined by a protocol (state machine) that it follows to do its part in processing a mission thread. Different OPFACs follow different protocols for different mission types.

An extensible, component-based design for FSATS follows directly from these observations. One component (`Vanilla`) defines an initial OPFAC class hierarchy and routines for sending and receiving messages, routing messages to appropriate missions, reading simulation scripts, and so forth. Figure 4 depicts the `Vanilla` component encapsulating multiple classes, one per OPFAC type. The OPFACs that are defined in `Vanilla` do not know how to react to external stimuli. Such reactions are encapsulated in mission components.

Each mission component encapsulates protocols (expressed as state machines) that are added to each OPFAC that could participate in a thread of this mission type. Composing a mission component with `Vanilla` extends each OPFAC with knowledge of how to react to particular external stimuli and how to coordinate its response with other OPFACs. For example, when the `WRFFE-artillery` component is added, a forward observer now has a protocol that tells it how to react when it sees an enemy tank—it creates a `WRFFE-artillery` message which it relays to its FIST. The FIST commander, in turn, follows his `WRFFE-artillery` protocol to forward this message to his battalion FSE, and so on. Figure 4 depicts the `WRFFE-artillery` component encapsulating multiple classes, again one per OPFAC type. Each enclosed class encapsulates a protocol which is added to its appropriate OPFAC class. Component composition is accomplished via inheritance, and is shown by dark vertical lines between class ovals in Figure 4. The same holds for other mission components (e.g., `TOT-artillery`). *Note that the classes that are instantiated are the bottom-most classes of these linear inheritance chains, because they embody all the protocols/features that have been grafted onto each OPFAC.* Readers will recognize this is an example of the GenVoca paradigm of Section 2.2, where components are mixin-layers.

The GenVoca-FSATS design has distinct advantages:

- it is mission-type *extensible* (i.e., it is comparatively easy to add new mission types to an existing GenVoca-FSATS simulator),[6]
- each mission type is defined largely independently of others, thereby reducing the difficulties of specification, coding, and debugging, and
- understandability is improved: OPFAC behavior is routinely understood and analyzed as mission threads. Mission-type components directly capture this simplicity, avoiding the complications of knowledge acquisition and engineering of rule sets.

Implementation. There are over twenty different mixin-layer components in GenVoca-FSATS, all of which are composed now to form a "fully-loaded" simulator. There are individual components for each mission type, just like Figure 4. However, there is no monolithic `Vanilla` component. We discovered that `Vanilla` could be decomposed into ten largely independent layers (totalling 97 classes) that deal with different aspects of the FSATS infrastructure. For example, there are distinct components for:

- OPFACs reading from simulation scripts,
- OPFAC communication with local and remote processes,
- OPFAC proxies (objects that are used to evaluate whether OPFAC commanders are supported by desired weapons platforms),
- different weapon OPFACs (e.g., distinct components for mortar, artillery, etc.), and
- GUI displays for graphical depiction of ongoing simulations.

Packaging these capabilities as distinct components, simplifies both specifications (because no extraneous details need to be included), and debugging (as components can largely be debugged in isolation). An important feature of our design is that all OPFACs are coded as threads executing within a single Java process. An "RMI adaptor" component transforms this design into a distributed program where each OPFAC thread executes in its own process at a different site [Batory et al. 1999]. The advantage here is that it is substantially easier to debug layers and mission threads within a single process than to debug remote

[6] Although a product-line of different FSATS simulators is possible; presently the emphasis of FSATS is on mission type extensibility. It is worth noting, however, that exponentially-large product-lines of FSATS simulators could be synthesized—i.e., if there are m mission components, there can be up to 2^m distinct compositions/simulators.

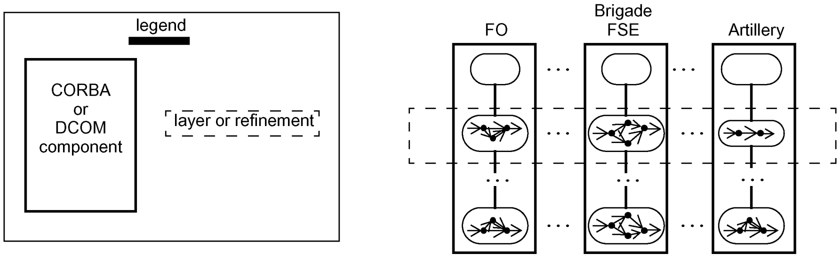


Fig. 5. CORBA and DCOM vs. layers (refinements).

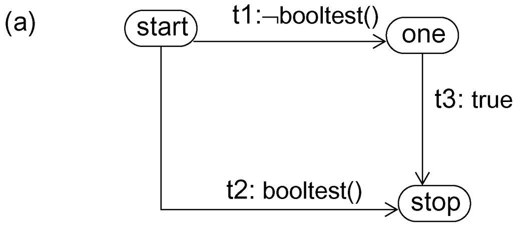
executions. Furthermore, the adaptor is included in an FSATS design only when distributed simulations are needed.

Perspective. It is worth comparing our notion of components with those that are common in today's software industry. Event-based distributed architectures, where DCOM and CORBA components communicate via message exchanges, is likely to be a dominant architectural paradigm of the future [Taylor 1999]. The original design of FSATS is a classic example: OPFAC programs are distributed DCOM/CORBA "components" that exchange messages. Yet the "components" common to distributed architectures are *orthogonal* to the components in the GenVoca-FSATS design. (This is depicted below in Figure 5 where each vertical inheritance chain corresponds to an OPFAC that is a CORBA or DCOM class, whereas an FSATS mission type is depicted by a horizontal slice through all OPFACs). That is, our components (layers) encapsulate fragments of *many* OPFACs, instead of encapsulating an *individual* OPFAC. (This is typical of approaches based on collaboration-based or "aspect-based" designs).

Event-based architectures are clearly extensible by their ability to add and remove component instances (e.g., adding and removing OPFACs from a simulation). This is (*OPFAC*) *object population* extensibility, which FSATS definitely requires. But FSATS also needs *feature* extensibility—OPFAC programs must be mission-type extensible. While these distinctions seem obvious in hind-sight, they were not so, prior to our work. FSATS clearly differentiates them.

3.2 A Domain-Specific Language for State Machines

We discovered that OPFAC rule sets were largely representations of state machines. We found that expressing OPFAC actions as state machines was a *substantial* improvement over rules; they are much easier to explain and understand, and require very little background to comprehend. A major goal of the redesign was to minimize the "conceptual distance" between architectural abstractions and their implementation. The problem we faced is that encodings of state machines are obscure, and given the situation that our specifications often *refined* previously created machines, expressing state machines in pure Java code was unattractive. To eliminate these problems, we used JTS to extend Java with a domain-specific language for declaring and refining state machines, so that our informal state machines (nodes, edges, etc.) had a direct expression as



(b)

```
state_machine exampleJavaSM {

   event_delivery receive_message(M m);          (1)
   on_error { error(-1,m); }                     (2)
   otherwise_default { ignore_message(m); }      (3)

   states start, one, stop;                      (4)

   edge t1 : start -> one                        (5)
   conditions !booltest()
   do { /* t1 action */ }

   edge t2 : start -> stop
   conditions booltest()
   do { /* t2 action */ }

   edge t3 : one -> stop
   conditions true
   do { /* t3 action */ }

   // methods and data members from here on... (6)

   boolean booltest() { ... }
   exampleJavaSM() { current_state = start; }
   ...
}
```

Fig. 6. JavaSM state machine specification.

a formal, compilable document. This extended version of Java is called *JavaSM*, and took us a bit more than a week to code into JTS.

Initial Declarations. A central idea of JavaSM is that a state machine specification translates into the definition of a single class. There is a generated variable (current_state) whose value indicates the current state of the protocol (i.e., state-machine-class instance). When a message is received by an OPFAC, a designated method is invoked with this message as an argument; depending on the state of the protocol, different transitions occur. Figure 6a shows a simple state machine with three states and three transitions. When a message arrives in the start state, if method booltest() is true, the state advances to stop; otherwise the next state is one.

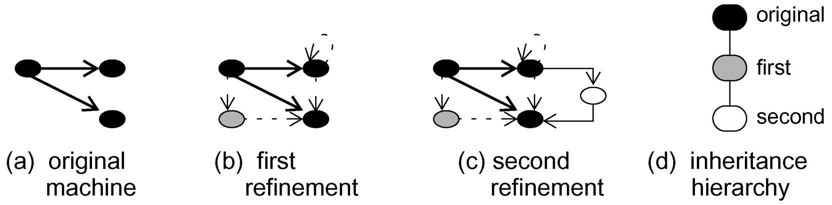


Fig. 7. Refining state machines.

Our model of FSATS required boolean conditions that triggered a transition to be arbitrary Java expressions with no side-effects, and the actions performed by a transition, to be arbitrary Java statements. Figure 6b shows a JavaSM specification of Figure 6a. (1) defines the name and formal parameters of the void method that delivers a message to the state machine. In the case that actions have corrupted the current state, (2) defines the code that is to be executed upon error discovery. When a message is received and no transition is activated, (3) defines the code that is to be executed (in this case, ignore the message). The three states in Figure 6a are declared in (4). Edges are declared in (5): each edge has a name, start state, end state, transition condition, and transition action. Java data member declarations and methods are introduced after edge declarations (6). When the specification of Figure 6b is translated, the class `exampleJavaSM` is generated. Additional capabilities of JavaSM are discussed in Batory et al. [1998].

Refinement Declarations. State machines can be progressively refined in a layered manner. A *refinement* is the addition of states, edges, data members, and methods to an existing machine. A common situation in FSATS is illustrated in Figure 7. Protocols for missions of the same general type (e.g., WRFFE) share the same protocol fragment for initialization (Figure 7a). A particular mission type (e.g, WRFFE-artillery) grafts on states and edges that are specific to it (Figure 7b). Additional missions contribute their own states and edges (Figure 7c), thus allowing complex state machines to be built in a stepwise manner.

The original state machine and each refinement are expressed as separate JavaSM specifications that are encapsulated in distinct layers. When these layers are composed, their JavaSM specifications are translated into a Java class hierarchy. Figure 7d shows this hierarchy: the root class was generated from the JavaSM specification of Figure 7a; its immediate subclass was generated from the JavaSM refinement specification of Figure 7b; the terminal subclass was generated from the JavaSM refinement specification of Figure 7c. Figure 8 sketches a JavaSM specification of this refinement chain.

Inheritance (i.e., class refinement) plays a central role in this implementation. All the states and edges in Figure 7a are inherited by the machine refinements of Figure 7b, and these states, edges, and so forth, are inherited by the machine refinements of Figure 7c. The machine that is executed, is created by instantiating the bottom-most class of the refinement chain of Figure 7d. Readers will again recognize this an example of the GenVoca paradigm of Section 2.2.

```
state_machine original {
   states one, two, three;

   edge a : one -> two ...
   edge b : one -> three ...
}

state_machine first refines original {
   states four;

   edge c : one -> four ...
   edge d : four -> three...
   edge e : two -> three ...
   edge f : two -> two ...
}

state_machine second refines first {
   states five;

   edge g : two-> five ...
   edge h : five -> three ...
}
```

Fig. 8. A JavaSM refinement hierarchy.

Perspective. Domain-specific languages for state machines are common (e.g., Berry and Gonthier 1992; Ellsberger et al. 1997; Harel 1987; Harel and Gery 1996; Neighbors 1997). Our way of expressing state machines—as states with enter and exit methods, edges with conditions and actions—is an elementary subset of Harel's Statecharts [Harel 1987; Harel and Gery 1996] and SDL extended finite state machines [Ellsberger et al. 1997]. The notion of refinement in Statecharts is the ability to explode individual nodes into complex state machines. This is very different than the notion of refinement explored in this paper. Our work is closer to the refinement of extended finite state machines in SDL where a process class (which encodes a state machine) can be refined via subclassing (i.e., new states and edges are added to extend the parent machine's capabilities). While the idea of state machine refinements is not new, it is new in the context of a DSL-addition to a general-purpose programming language (Java), and it is fundamental in the context of component-based development of FSATS simulators.

4. PRELIMINARY RESULTS

Our preliminary findings are encouraging—the objectives of the redesign were met by the GenVoca-FSATs design:

- it is now possible to specify, add, verify, and test a mission type independent of other mission types (because a version of FSATS can be created with a single mission),

- it is now possible to remove and replace mission types to accommodate varying user requirements, and
- JavaSM allows a direct implementation of a specification, thereby reducing the "conceptual distance" between specification and implementation.

As is common in re-engineering projects, detailed statistics on the effort involved in the original implementation are not available. However, we can make some rough comparisons. From our experience with the original FSATS simulator, we estimate the time to add a mission to be about 1 month. A similar addition to GenVoca-FSATS, including one iteration to identify and correct an initial misunderstanding of the protocols for that mission, was accomplished in about 3 days.

To evaluate the redesign in a less anecdotal fashion, we collected statistics on program complexity. We used simple measures of class complexity: the number of methods (**nmeth**), the number of lines of code (**nloc**), and the number of tokens/symbols (**nsymb**) per class. (We originally used other metrics [Chidamber and Kemerer 1991], but found they provided no further insights.) Because of our use of JTS, we have access to both component-specification code (i.e., layered JavaSM code written by FSATS engineers), and generated non-layered pure-Java code (which approximates code that would have been written by hand). By using metrics to compare pure-Java code vs. JavaSM code, and layered vs. non-layered code, we can quantitatively evaluate the impact of layering and JavaSM on reducing program complexity, a key goal of our redesign.

Complexity of Non-Layered Java Code. Consider a non-layered design of FSATS. Suppose all of our class refinement chains were "squashed" into single classes; these would be the classes that would be written by hand if a non-layered design were used. Consider the FSATS class hierarchy that is rooted by class `MissionImpl`; this class encapsulates methods and an encoding of a state machine that is shared by all OPFACS. (In our prototype, we implemented different variants of WRFFE missions.) Class `FoMission`, a subclass of `MissionImpl`, encapsulates the additional methods and the Java-equivalent of state machine edges/states that define the actions that are specific to a Forward Observer. Other subclasses of `MissionImpl` encapsulate additions that are specific to other OPFACs. The "Pure Java" columns of Table I present complexity statistics of the `FoMission` and `MissionImpl` classes. *Note that our statistics for subclasses, by definition, must be no less than those of their superclasses* (*because the complexity of superclasses is inherited*).

One observation is immediately apparent: the number of methods (117) in `MissionImpl` is huge. Different encoding techniques for state machines might reduce the number, but the complexity would be shifted elsewhere (e.g., methods would become more complicated). Because our prototype only deals with WRFFE missions, we must expect that the number of methods in `MissionImpl` will increase, as more mission types are added. Consider the following: there are 30 methods in class `MissionImpl` alone that are used in WRFFE missions. When we add a WRFFE mission that is specialized for a particular weapon system (e.g., mortar), another 10 methods are added. Since WRFFE is representative of mission complexity, as more mission types are added with their

Table I. Statistics for Non-Layered Implementation of Class FoMission

	Pure Java			JavaSM		
Class Name	**nmeth**	**nloc**	**nsymb**	**nmeth**	**nloc**	**nsymb**
`MissionImpl`	117	461	3452	54	133	1445
`FoMission`	119	490	3737	56	143	1615

weapon specializations, it is not inconceivable that `MissionImpl` will have several hundred methods. Clearly, such a class would be both incomprehensible and unmaintainable.[7]

Now consider the effects of using JavaSM. The "JavaSM" columns of Table I show corresponding statistics, where state exit and enter declarations and edge declarations are treated as (equivalent in complexity to) method declarations. We call such declarations *method-equivalents*. Comparing the corresponding columns in Table I, we see that coding in JavaSM reduces software complexity by a factor of 2. That is, the number of method-equivalents is reduced by a factor of 2 (from 119 to 56), the number of lines of code is reduced by a factor of 3 (from 490 to 143), and the number of symbols is reduced by a factor of 2 (from 3737 to 1615). However, the problem that we noted in the pure-Java implementation remains. Namely, the generic WRFFE mission contributes over 10 method-equivalents to `MissionImpl` alone; when WRFFE is specialized for a particular weapon system (e.g., mortar), another 3 method-equivalents are added. While this is substantially better than its non-layered pure-Java equivalent, it is not inconceivable that `MissionImpl` will have over a hundred method-equivalents in the future. *While the JavaSM DSL indeed simplifies specifications*, *it only delays the onset of design fatigue. Non-layered designs of FSATS may be difficult to scale and ultimately hard to maintain*, *even if the JavaSM DSL is used.*

Complexity of Layered Java Code. Now consider a layered design implemented in pure Java. The "Inherited Complexity" columns of Table II show the inheritance-cumulative statistics for each class of the `MissionImpl` and `FoMission` refinement chains. The rows where `MissionImpl` and `FoMission` data are listed in **bold** represent classes that are the terminals of their respective refinement chains. These rows correspond to the rows in Table I. The "Isolated Complexity" columns of Table II show complexity statistics for individual classes of Table II (i.e., we are measuring class complexity, and not including the complexity of superclasses). Note that most classes are rather simple. The `MissionAnyL.MissionImpl` class, for example, is the most complex, with 43 methods. (This class encapsulates "infrastructure" methods used by all missions.) Table II indicates that layering disentangles the logic of different features of the `FoMission` and `MissionImpl` classes into units that are small enough to be comprehensible and manageable by programmers. For example,

[7] It would be expected that programmers would introduce some other modularity, thereby decomposing a class with hundreds of methods into multiple classes with smaller numbers of methods. While this would indeed work, it would complicate the "white-board"-to-implementation mapping (which is what we want to avoid) and there would be no guarantee that the resulting design would be mission-type extensible.

Table II. Statistics for a Layered Java Implementation of Class FoMission

	Inherited Complexity			Isolated Complexity		
Class Name	**nmeth**	**nloc**	**nsymb**	**nmeth**	**nloc**	**nsymb**
`MissionL.MissionImpl`	9	25	209	9	25	209
`ProxyL.MissionImpl`	11	30	261	2	5	52
`MissionAnyL.MissionImpl`	51	179	1431	43	149	1170
`MissionWrffeL.MissionImpl`	83	314	2342	35	135	911
`MissionWrffeMortarL.MissionImpl`	93	358	2677	13	44	335
`MissionWrffeArtyL.MissionImpl`	109	425	3187	19	67	510
`MissionWrffeMlrsL.MissionImpl`	**117**	**461**	**3452**	**11**	**36**	**265**
`BasicL.FoMission`	117	461	3468	0	0	16
`MissionWrffeMortarL.FoMission`	117	468	3547	4	7	79
`MissionWrffeArtyL.FoMission`	119	484	3687	7	16	140
`MissionWrffeMlrs.FoMission`	**119**	**490**	**3737**	**3**	**6**	**50**

Table III. Statistics on a Layered JavaSM Implementation of Class FoMission

	Inherited Complexity			Isolated Complexity		
Class Name	**nmeth**	**nloc**	**nsymb**	**nmeth**	**nloc**	**nsymb**
`MissionL.MissionImpl`	8	20	169	8	20	169
`ProxyL.MissionImpl`	10	25	221	2	5	52
`MissionAnyL.MissionImpl`	34	90	877	24	65	656
`MissionWrffeL.MissionImpl`	45	115	1132	11	25	255
`MissionWrffeMortarL.MissionImpl`	48	121	1231	3	6	99
`MissionWrffeArtyL.MissionImpl`	52	129	1383	4	8	152
`MissionWrffeMlrsL.MissionImpl`	**54**	**133**	**1445**	**2**	**4**	**62**
`BasicL.FoMission`	54	133	1461	0	0	16
`MissionWrffeMortarL.FoMission`	54	136	1518	2	3	57
`MissionWrffeArtyL.FoMission`	55	140	1586	3	4	68
`MissionWrffeMlrs.FoMission`	**56**	**143**	**1615**	**2**	**3**	**29**

instead of having to understand a class with 117 methods, the largest layered subclass has 43 methods; instead of 461 lines of code there are 149 lines, and so on.

To gauge the impact of a layered design in JavaSM, consider the "Inherited Complexity" columns of Table III that show statistics for `MissionImpl` and `FoMission` refinement chains written in JavaSM. The "Isolated Complexity" columns of Table III show corresponding statistics for individual classes. They show that layered JavaSM specifications are indeed compact: instead of a class with 43 methods there are 24 method-equivalents, instead of 149 lines of code there are 65 lines, and so on. Thus, a combination of domain-specific languages and layered designs greatly reduces program complexity.

Our use of the "Isolated Complexity" metric as the indicator of class complexity requires some discussion. It is indeed the case that the "true" complexity of a class is somehow related to the total complexity of its superclasses plus the additional complexity of the class itself. So it could be argued that the "Inherited Complexity" metric might be a better measure of the actual difficulty of understanding a given layer. This is not the case for FSATS. Typically FSATS layers simply invoke methods of their superclass, much in the same way that COM and CORBA components invoke methods of server interfaces. Implementation details are hidden behind such interfaces, thereby making it easy for programmers to invoke server methods without having to know how servers are implemented. The same holds for layers in FSATS. The only difference here, is that a *few* methods of each FSATS class override (i.e., extend) previously defined methods, thereby requiring programmers to know more of the "guts" of superclass implementation. But for FSATs (and other generators that we have built), this additional implementation knowledge is minimal. Further, there may be layers in superclass implementations that provide infrastructure that programmers of mission-layers do not need to be aware of at all; they are simply methods that are private to that layer. For these reasons, we believe that "Isolated Complexity" is closer to the true complexity of a class than "Inherited Complexity."

The reduction in program complexity is a key goal of our project; these tables support the observations of FSATS engineers: the mapping between a "white-board" design of FSATS protocols and an implementation, is both direct and invertible with layered JavaSM specifications. That is, writing components in JavaSM matches the informal designs that domain experts use; it requires fewer mental transformations from design to implementation, which simplifies maintenance and extensibility, and makes for a much less error-prone product. In contrast, mapping from the original FSATS implementation back to the design was not possible due to the lack of an association of any particular rule or set of rules with a specific mission.

5. CONCLUSIONS

Extensibility is the property that simple changes to the design of a software artifact require a proportionally simple effort to modify its source code. Extensibility is a result of premeditated engineering, whereby anticipated variabilities in a domain are made simple by design. Two complementary technologies are emerging that make extensibility possible: *product-line architectures* (*PLAs*) and *domain-specific languages* (*DSLs*). Product-lines rely on components to encapsulate the implementation of basic features or "aspects" that are common to applications in a domain; applications are extensible through the addition and removal of components. Domain-specific languages enable applications to be programmed in domain abstractions, thereby allowing compact, clear, and machine-processable specifications to replace detailed and abstruse code. Extensibility is achieved through the evolution of specifications.

FSATS is a simulator for Army fire support and is representative of a complex domain of distributed command-and-control applications. The original implementation of FSATS had reached a state of design fatigue, where anticipated changes/enhancements to its capabilities would be expensive to realize. We undertook the task of redesigning FSATS so that its inherent and projected variabilities—that of adding new mission types—would be easy to introduce. Another important goal was to minimize the "conceptual distance" from "white-board" designs of domain experts to actual program specifications; because of the complexity of fire-support, the specifications had to closely match these designs to make the next-generation FSATS source understandable and maintainable.

We achieved the goals of extensibility and understandability through an integration of PLA and DSL technologies. We used a GenVoca PLA to express the building blocks of fire support simulators as layers or refinements, whose addition or removal simultaneously impacts the source code of multiple, distributed programs. But a layered design was insufficient, because our components could not be easily written in pure Java. The reason is that the code expressing state machine abstractions was so low-level that it would be difficult to read and maintain. We addressed this problem by extending the Java language with a domain-specific language, to express state machines and their refinements, and wrote our components in this extended language. Preliminary findings confirm that our component specifications are substantially simplified; "white-board" designs of domain experts have direct and invertible expressions in our specifications. Thus, we believe that the combination of PLAs and DSLs is essential in creating extensible fire support simulators.

While fire support is admittedly a domain with specific and unusual requirements, there is nothing domain-specific about the need for PLAs, DSLs, and their benefits. In this regard, FSATS is not unusual; it is a classical example of the many domains where both technologies naturally complement each other to produce a result that is better than either technology could deliver in isolation. Research on PLA and DSL technologies should focus on infrastructures (such as IP [Simonyi 1995] and JTS [Batory et al. 1998]) that support their integration; research on PLA and DSL methodologies must be more cognizant that synergy is not only possible, but desirable.

ACKNOWLEDGMENTS

We thank Dewayne Perry (UTexas) for his insightful comments on an earlier draft and bringing SDL to our attention, and Frank Weil (Motorola) for clarifying discussions on SDL state machines. We also thank the referees for their helpful suggestions that improved the final draft of this paper.

REFERENCES

FSATS 1999. "System Segment Specification (SSS) for the Fire Support Automated Test System", Applied Research Laboratories, The University of Texas. See URL http://www.arlut.utexas.edu/~fsatswww/fsats.shtml.

BATORY, D. AND O'MALLEY, S. 1992. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. Softw. Eng. Methodol.* (Oct.).

BATORY, D., COGLIANESE, L., GOODWILL, M., AND SHAFER, S. 1995. Creating Reference Architectures: An Example from Avionics. *Symposium on Software Reusability*, Seattle, WA. (Apr.).

BATORY, D., LOFASO, B., AND SMARAGDAKIS, Y. 1998. JTS: Tools for Implementing Domain-Specific Languages. *5th International Conference on Software Reuse*, Victoria, Canada (June). http://www.cs.utexas.edu/users/schwartz/JTS30Beta2.htm.

BATORY, D. AND GERACI, B. J. 1997. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Trans. Softw. Eng.* (Feb.), 67–82.

BATORY, D. 1998. Product-Line Architectures. *Smalltalk and Java Conference*, Erfurt, Germany (Oct.).

BATORY, D., SMARAGDAKIS, Y., AND COGLIANESE, L. 1999. Architectural Styles as Adaptors. *Software Architecture*, Patrick Donohoe, ed., Kluwer Academic Publishers.

BATORY, D., CHEN, G., ROBERTSON, E., AND WANG, T. 2000. Design Wizards and Visual Programming Environments for GenVoca Generators. *IEEE Trans. Softw. Eng.* (May), 441–452.

BAXTER, I. 1992. Design Maintenance Systems. *CACM* (Apr.).

BERRY, G. AND GONTHIER, G 1992. The Esterel Synchronous Programming language: Design, Semantics, and Implementation. *Science of Computer Programming*. 87–152.

BOSCH, J. 1999. Product-Line Architectures in Industry: A Case Study. *ICSE*, Los Angeles, CA.

CARDONE, R. AND LIN, C. 2001. Comparing Frameworks and Layered Refinement. ICSE Toronto.

CHIDAMBER S. R. AND KEMERER, C. F. 1991. Towards a Metrics Suite for Object Oriented Design. *OOPSLA*.

CZARNECKI, K. AND EISENECKER, U. W. 1999. Components and Generative Programming. *ACM SIGSOFT*.

VAN DEURSEN, A. AND KLINT, P. 1997. Little Languages: Little Maintenance? *SIGPLAN Workshop on Domain-Specific Languages*.

EICK, S. G., GRAVES, T. L., KARR, A. F., MARRON, J. S., AND MOCKUS, A. 2001. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Trans. Softw. Eng.*, *27*:1 (January), 1–12.

ELLSBERGER, J., HOGREFE, D., AND SARMA, A. 1997. *Formal Object-Oriented Language for Communicating Systems*. Prentice-Hall, Englewood Cliffs, NJ.

FINDLER, R. B. AND FLATT, M. "Modular Object-Oriented Programming with Units and Mixins", *ICFP 98*.

GAMMA E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts.

GOGUEN, J. A. 1986. Reusing and Interconnecting Software Components. *IEEE Computer* (Feb.).

GOMAA, H., KERSCHBERG, L., AND SUGAMARAN, V. 1992. A Knowledge-Based Approach to Generating Target System Specifications from a Domain Model. *IFIP Congress 1*, 252–258.

GRISS, M. 2000. Implementing Product-Line Features by Composing Component Aspects. *First International Software Product-Line Conference*, Denver, CO. (Aug.).

HAREL, D. 1987. Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, 231–274.

HAREL, D. AND GERY, E. 1996. Executable Object Modeling with Statecharts. *ICSE*.

KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. 1990. Feature-Oriented Domain Analysis Feasibility Study, SEI. Technical Report CMU/SEI-90-TR-21 (Nov.).

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J., AND IRWIN, J. 1997. Aspect-Oriented Programming, *ECOOP*. 220–242.

LOPEZ-HERREJON, R. E. AND BATORY, D. 2001. A Standard Problem for Evaluating Product-Line Methodologies. *Third International Conference on Generative and Component-Based Software Engineering* (*GCSE 2001*), (Sept. 9–13), Messe Erfurt, Erfurt, Germany.

MAGNAVOX, 1999. System Segment Specification (SSS) for the Advanced Field Artillery Tactical Data System (AFATDS).

NEIGHBORS, J. 1989. Draco: A Method for Engineering Reusable Software Components, in T. J. Biggerstaff and A. Perlis, eds., *Software Reusability*, Addison-Wesley/ACM Press.

NEIGHBORS, J. 1997. "DataXfer Protocol," BayFront Technologies. 1997, URL `http://bayfront-technologies.com`.

REENSKAUG, T., ET AL. 1992. "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems", *Journal of Object-Oriented Programming*, 5(6) (Oct.), 27–41.

RICH, C. AND WATERS, R. C. 1990. *The Programmer's Apprentice*, ACM Press.

Software Engineering Institute 2001. The Product Line Practice (PLP) Initiative, URL `http://www.sei.cmu.edu/plp/plp_init.html`.

SIMONYI, C. 1995. The Death of Computer Languages, the Birth of Intentional Programming, *NATO Science Committee Conference*.

SMARAGDAKIS, Y. AND BATORY, D. 1998. Implementing Layered Designs with Mixin Layers, *ECOOP*.

SMARAGDAKIS, Y. AND BATORY, D. 2002. Mixin Layers: An Object-Orinted Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Trans. Softw. Eng.* Method.

TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, S. M. JR. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns, *ICSE*.

Taylor, R. 1999. Panel on Software Reuse. *Motorola Software Engineering Symposium*, Ft. Lauderdale.

Tokuda, L. and Batory, D. 1999. Evolving Object-Oriented Architectures with Refactorings. *Conference on Automated Software Engineering*, Orlando, FL.

Van Hilst, M. and Notkin, D. 1996. Using Role Components to Implement Collaboration-Based Designs, *OOPSLA*, 359–369.

Weiss, D. M. and Lai, C. T. R 1999. *Software Product-Line Engineering*. Addison-Wesley.

Received February 2001; revised September 2001; accepted October 2001

Accepted for publication, ACM Transactions on □
Software Engineering and Methodology.

Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs

YANNIS SMARAGDAKIS
Georgia Institute of Technology
and
DON BATORY
The University of Texas at Austin

A “refinement” is a functionality addition to a software project that can affect multiple dispersed implementation entities (functions, classes, etc.). In this paper, we examine large-scale refinements in terms of a fundamental object-oriented technique called collaboration-based design. We explain how collaborations can be expressed in existing programming languages or be supported with new language constructs (which we have implemented as extensions to the Java language). We present a specific expression of large-scale refinements called *mixin layers*, and demonstrate how it overcomes the scalability difficulties that plagued prior work. We also show how we used mixin layers as the primary implementation technique for building an extensible Java compiler, JTS.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures; D.2.13 [**Software Engineering**]: Reusable Software; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Design, Languages

Additional Key Words and Phrases: Collaboration-based design, component-based software, product-line architectures

1. INTRODUCTION

The history of software design and programming languages intimately evolves around the concept of modularity. Modules encapsulate primitive functionality or services that, ideally, can be reused in the construction of many applications. The granularity of modules has evolved from small scale, to medium scale, and now to large-scale: that is, from functions, to abstract data types or classes (i.e., suites of interrelated functions), and now more commonly to components or packages (i.e.,

The authors gratefully acknowledge funding by the University of Texas Applied Research Labs and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

Authors' addresses: Yannis Smaragdakis, College of Computing, Georgia Tech, Atlanta, GA 30332; Don Batory, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712.

suites of interrelated classes). The benefit of increased module scale is that of economics—applications are easier to build from fewer and larger parts—and design simplicity—applications are easier to comprehend when modules encapsulate, and thus hide, irrelevant implementation details.

The benefits of scaled modularity, however, are driven by reuse. The more a module is reused, the more valuable it becomes. But there is an ironic twist: the larger the module, the more specific its use and functionality, and this, in turn, reduces the likelihood that other applications will need its exact capabilities. In other words, it seems that reuse opportunities become fewer as a module becomes larger: scaling modularity seems to defeat the purpose of reuse, and this is exactly the opposite of what we want [Biggerstaff 1994].

The solution to this problem lies in a very different concept of modularity, where neither entire functions, entire classes, or entire packages are the answer. Instead, the unit of modularity that we seek encapsulates *fragments* of multiple classes, which in turn encapsulate *fragments* of multiple functions. An extensive body of research has shown that such units are indeed the reusable building blocks of large-scale modules: composing sets of class fragments yields a package of fully-formed classes. This recognition has become particularly clear in the area of software product-lines, where the goal is to construct large families of related applications from primitive and reusable components. The components that made this possible encapsulated fragments of classes.

We use the term *refinement* (also in [Batory and Geraci 1997]) for any such unit of functionality in a software system. A refinement is a functionality addition to a program that introduces a conceptually new service, capability, or feature, and may affect multiple implementation entities. Various researchers have offered different descriptions, implementations, and names to fairly analogous concepts over the years, including layers [Batory et al. 1988], collaborations [Reenskaug et al. 1992], subjects [Ossher and Harrison 1992; Tarr et al. 1999] and aspects [Kiczales et al. 1997]. Parnas's classic work [1979] has offered much of the software engineering context for these approaches.[1]

We believe that a prominent characteristic of successful refinement technologies is scalability. Implementing microscopic refinements (i.e., refinements that dealt with code fragments at the expression level) has not produced great software engineering advances in the past and is unlikely to do so in the future. The novelty of current research strikes at the core problem—that of scaling the unit of refinement from a

[1] The definition of "refinement" that seems closest to our intended meaning is "the act of making improvement by introducing subtleties or distinctions" (Merriam-Webster's Dictionary). Formal approaches to programming use the term "refinement" to denote the elaboration of a program by adding more implementation detail until a fully concrete implementation is reached. The set of behaviors (i.e., the legal variable assignments) of a "refined" program is a subset of the behaviors of the original "unrefined" program. This appears to be different from our use of the term. Our "refinements" follow the dictionary definition by adding "subtleties or distinctions" at the *design* level. At the implementation level, however, a refinement can yield dramatic changes: both the exported functionality (semantics of operations) and the exported interface (signatures of operations) may change. Thus, unlike the use of "refinement" in formal approaches to programming, the set of allowed behaviors of our "refined" program might not be a subset of the behaviors of the "unrefined" program.

microscopic scale to large scale, where a single refinement alters multiple classes of an application. A large-scale refinement exhibits “cross-cutting”—multiple classes must be updated simultaneously and consistently. Thus, composing a few large scale refinements yields an entire application. This means that the inverse relationship between module size and reusability that has crippled conventional concepts no longer applies, and a fresh look at software modularity has become a topic of wide-spread interest.

This paper is about modular implementations of large-scale refinements and the development of families of related applications through refinement. In particular, we show that a fundamental object-oriented concept, called *collaboration-based* designs, is in fact how large-scale refinements are expressed in object-oriented models. We begin by explaining the core ideas of collaboration-based design and how they are related to large-scale refinements. We then show how these ideas can be expressed in existing programming languages, or supported with new language constructs (which we have implemented as extensions of the Java language). In particular, we introduce a specific expression of large-scale refinements called *mixin layers*, and demonstrate how it extends and overcomes problems of prior work on the refinement-based designs of VanHilst and Notkin [1996b; 1996c; 1996a; 1997] and application frameworks [Johnson and Foote 1988]. Mixin layers implementations are discussed, but our paper intends to convince the reader that one *should* implement programs using mixin layers, not that one *is merely able to* do so. Better implementations than the ones we propose may be possible, or languages other than the ones we examine may offer more complete support for mixin layers. In either case, this would be independent from our main argument which is one of desirability of application development through mixin layers. As a practical validation, we show how we used mixin layers as the primary implementation technique in a medium-size project: the JTS tool suite for implementing domain-specific languages. Our experience shows that the mechanism is versatile and can handle refinements of substantial size.

2. BACKGROUND: COLLABORATION BASED DESIGNS

Collaboration-based or *role-based* designs have been the subject of many papers [Cunningham and Beck 1989; Helm et al. 1990; Holland 1992; Reenskaug et al. 1992; VanHilst and Notkin 1996b]. The concept may have originated with Reenskaug, et al. [1992] but the ideas have been used in various forms, often without being named (e.g., [Batory et al. 1988]). A good introduction to collaboration-based design can be found in the presentation of the OORAM approach [Reenskaug et al. 1992]. A detailed treatment of collaboration-based designs, together with a discussion of how to derive them from use-case scenarios [Rumbaugh 1994] can be found in VanHilst's Ph.D. dissertation [1997].

2.1 Collaborations and Roles

In an object-oriented design, objects are encapsulated entities but are rarely self-sufficient. Although an object is fully responsible for maintaining the data it encapsulates, it needs to cooperate with other objects to complete a task. An interesting way to encode object interdependencies is through collaborations. A *collaboration* is a set of objects and a protocol (i.e., a set of allowed behaviors) that determines how these objects interact. The part of an object enforcing the protocol that a collaboration prescribes is called the object's *role* in the collaboration. Objects of an application generally participate in multiple collaborations simultaneously and, thus, may encode several distinct roles. Each collaboration, in turn, is a collection of roles, and represents relationships across corresponding objects. Essentially, a role isolates the part of an object that is relevant to a collaboration from the rest of the object. Different objects can participate in a collaboration, as long as they support the required roles.

In collaboration-based design, the objective is to express an application as a composition of largely independently-definable collaborations. *Viewed in terms of design modularity, collaboration-based design acknowledges that a unit of functionality (module) is neither a whole object nor a part of it, but can cross-cut several different objects.* If a collaboration is reasonably independent of other collaborations (i.e., a good approximation of an ideal module) the benefits are great. First, the collaboration can be reused in a variety of circumstances where the same functionality is needed, by just mapping its roles to the right objects. Second, any changes in the encapsulated functionality will only affect the collaboration and will not propagate throughout the whole application.

In abstract terms, a collaboration is a view of an object-oriented design from the perspective of a single concern, service, or feature. For instance, a collaboration can be used to express a producer-consumer relationship between two communicating objects. Clearly, this collaboration prescribes roles for (at least) two objects and there is a well-defined "protocol" for their interactions. Interestingly, the same collaboration could be instantiated more than once in a single object-oriented design, with the same objects playing different roles in every instantiation. In the example of the producer-consumer collaboration, a single object could be both a producer (from the perspective of one collaboration) and a consumer (from the perspective of another).

Figure 1 depicts the overlay of objects and collaborations in an abstract application involving three different objects (*OA*, *OB*, *OC*), each supporting multiple roles. Object *OB*, for example, encapsulates four distinct roles: `B1`, `B2`, `B3`, and `B4`. Four different collaborations (*c1*, *c2*, *c3*, *c4*) capture distinct aspects of the application's functionality. Each collaboration prescribes roles to certain objects. For example, collaboration *c2* contains two distinct roles, `A2` and `B2`, which are assumed by distinct objects (namely *OA* and *OB*). An object does not need to play a role in every collaboration—for instance, *c2* does not affect object *OC*.

Collaborations can be composed dynamically at application run-time or statically at application compile-time. In this paper, we examine the static composition of collaborations, where roles that are played by an object are uniquely determined by its class. For instance, in Figure 1, all three objects must belong to different classes (since they all support different sets of roles). The work described in this paper can be generalized to dynamic compositions of collaborations.

From a broader perspective, a collaboration is a large-scale refinement. Again, a refinement elaborates a program to extend its functionality or to add implementation details. A refinement is large scale if it modifies multiple classes of an application. For example, when collaboration *c4* is (statically) added to the pro-

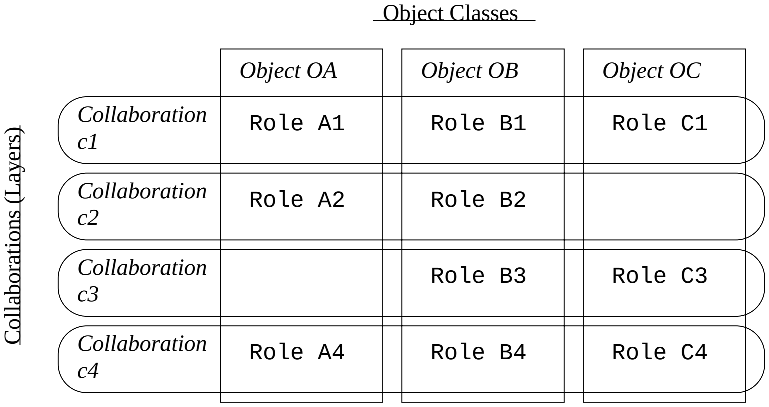


Fig. 1. Example collaboration decomposition. Ovals represent collaborations, rectangles represent objects, their intersections represent roles.

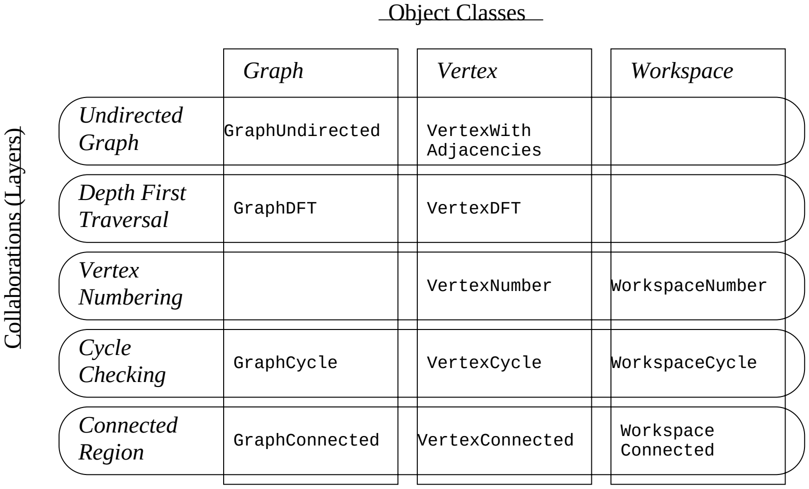
gram of Figure 1, the classes for objects OA, OB, and OC are updated consistently and simultaneously so that the "feature" or "service" defined by *c4* is appropriately implemented. Thus, composing collaborations is an example of refinement, where a simple program is progressively elaborated into a more complex one. Collaborations are large-scale and reusable refinements—they can be used in the construction of many programs.

2.2 An Example

As a running example that illustrates important points of our discussion, we consider a graph traversal application that was examined initially by Holland [1992] and subsequently by VanHilst and Notkin [1996b]. Doing so affords not only a historical perspective on the development of collaboration-based designs, but also a perspective on the contribution of this work. The application defines three different operations (algorithms) on an undirected graph, all based on depth-first traversal: *Vertex Numbering* numbers all nodes in the graph in depth-first order, *Cycle Checking* examines whether the graph is cyclic, and *Connected Regions* classifies graph nodes into connected graph regions. That is, a client of this application can instantiate a graph and separately invoke algorithms that perform vertex numbering, cycle checking, and/or find connected regions on a graph. The application itself has three distinct classes: *Graph*, *Vertex*, and *Workspace*. The *Graph* class describes a container of nodes with the usual graph properties. Each node is an instance of the *Vertex* class. Finally, the *Workspace* class includes the application part that is specific to each graph operation. For example, the *Workspace* object for a *Vertex Numbering* operation holds the value of the last number assigned to a vertex as well as the methods to update this number.

In decomposing this application into collaborations, we need to capture distinct aspects as separate collaborations. A decomposition of this kind is straightforward and results in five distinct collaborations.

One collaboration (*Undirected Graph*) encapsulates properties of an undirected



	Graph	*Vertex*	*Workspace*
Undirected Graph	GraphUndirected	VertexWith Adjacencies	
Depth First Traversal	GraphDFT	VertexDFT	
Vertex Numbering		VertexNumber	WorkspaceNumber
Cycle Checking	GraphCycle	VertexCycle	WorkspaceCycle
Connected Region	GraphConnected	VertexConnected	Workspace Connected

Fig. 2. Collaboration decomposition of the example application domain: A depth-first traversal of an undirected graph is specialized to yield three different graph operations. Ovals represent collaborations, rectangles represent classes.

graph. This is clearly an independent aspect of the application—the problem could very well be defined for directed graphs, for trees, etc.

Another collaboration (*Depth First Traversal*) encapsulates the specifics of depth-first traversals and provides a clean interface for extending traversals. That is, at appropriate moments during a traversal (the first time a node is visited, when an edge is followed, and when a subtree rooted at a node is completely processed) control is transferred to specialization methods that can obtain information from the traversal collaboration and supply information to it. Consider the *Vertex Numbering* operation as a refinement of a depth-first traversal. Numbering is realized by specializing the action when visiting a node for the first time during a traversal. The action assigns a number to the node and increases the count of visited nodes.

Using this approach, each of the three graph operations can be seen as a refinement of a depth-first traversal and each can be expressed by a single collaboration. Figure 2 is reproduced from [VanHilst and Notkin 1996b] and presents the collaborations and classes of our example application domain. The intersection of a class and a collaboration in Figure 2 represents the role prescribed for that class by the collaboration. A role encodes the part of an object that is specific to a collaboration. For instance, the role of a *Graph* object in the "*Undirected Graph*" collaboration supports storing and retrieving a set of vertices. The role of the same object in the "*Depth First Traversal*" collaboration implements a part of the depth-first traversal algorithm. (In particular, it contains a method that initially marks all vertices of a graph *not-visited* and then calls the method for depth-first traversal on each graph vertex object.)

The goal of a collaboration-based design is to encapsulate within a collaboration all dependencies between classes that are specific to a particular service or feature. In this way, collaborations themselves have no outside dependencies and can be reused in a variety of circumstances. The "*Undirected Graph*" collaboration, for instance, encodes the properties of an undirected graph (pertaining to the *Graph* and *Vertex* classes, as well as the interactions between objects of the two). Thus, it can be reused in any application that deals with undirected graphs. Ideally, if we could define an "interface" to a collaboration, we should also be able to easily replace one collaboration with another that exports the same interface. For instance, it would be straightforward to replace the "*Undirected Graph*" collaboration with one representing a directed graph, assuming that both collaborations exported the same interface.

Of course, simple interface conformance does not guarantee composition correctness—the application writer must ensure that the algorithms used (for example, the depth-first traversal) are still applicable after the change. The algorithms presented by Holland [1992] for this example are, in fact, general enough to be applicable to a directed graph. If, however, a more efficient, specialized-for-undirected-graphs algorithm was used (as is, for instance, possible for the *Cycle Checking* operation) the change would yield incorrect results. [Smaragdakis 1999; Smaragdakis and Batory 1998; Batory and Geraci 1997] discuss in detail the issue of ensuring that collaborations are actually interchangeable.

Although we have focussed on a single application that supports all three graph operations, it is easy to see how variants of this application could be created (e.g., by omitting some operations or adding new operations), where each variant would be described by the use of different collaborations. This very fact makes collaboration-based designs ideal for describing product-line architectures, that is, designs for families of related applications. As we will see, collaborations define the building blocks for application families, compositions of these building blocks yields different product-line members.

3. IMPLEMENTING COLLABORATION-BASED DESIGNS WITH MIXIN LAYERS

3.1 Mixin Classes and Mixin Layers

A refinement of an object-oriented class is encapsulated by a subclass: a subclass can add new methods and data members, as well as override existing methods of its superclass. Thus, inheritance is a built-in mechanism for statically refining classes in object-oriented languages. The challenge is to scale inheritance from refining individual classes to expressing the large-scale refinements of collaboration-based designs.

A solution is to build on an existing object-oriented construct called a *mixin*. Mixins are similar to classes but with some added flexibility, as described shortly. Unfortunately, mixins alone are not sufficient to express large-scale refinements—they suffer from only being able to refine a single class at a time and not a collection of cooperating classes. To address this, we introduce *mixin-layers*: a scaled-up form of mixins that can contain multiple smaller mixins.

3.1.1 *Introduction to Mixins.* The term *mixin class* (or just "mixin") has been overloaded to mean several specific programming techniques and a general mechanism that they all approximate. Mixins were originally explored in the context of the Lisp language with object-systems like Flavors [Moon 1986] and CLOS [Kiczales et al. 1991]. They were defined as classes that allow their superclass to be determined by *linearization* of multiple inheritance. In C++, the term has been used to describe classes in a particular (multiple) inheritance arrangement: as superclasses of a single class that themselves have a common *virtual base class* (see [Stroustrup 1997], p.402). Both of these mechanisms are approximations of a general concept described by Bracha and Cook [1990], and here we use "mixin" in this general sense.

The main idea of mixins is simple: in object-oriented languages, a superclass can be defined without specifying its subclasses. This property is not, however, symmetric: when a subclass is defined, it must have a specific superclass. Mixins (also commonly known as *abstract subclasses* [Bracha and Cook 1990]) represent a mechanism for specifying classes that eventually inherit from a superclass. This superclass, however, is not specified at the site of the mixin's definition. Thus a single mixin can be instantiated with different superclasses yielding widely varying classes. This property makes them appropriate for defining uniform incremental extensions for a multitude of classes. When a mixin is instantiated with one of these classes as a superclass, it produces a class incremented with the additional behavior.

Mixins can be implemented using parameterized inheritance: it is a class whose superclass is specified by a parameter. Using C++ syntax we can write a mixin as:

```
template <class Super> class Mixin : public Super {
  ... /* mixin body */
};
```

Mixins are flexible and can be applied in many circumstances without modification. To give an example, consider a mixin implementing *operation counting* for a graph. Operation counting means keeping track of how many nodes and edges have been visited during the execution of a graph algorithm. (This simple example is one of the non-algorithmic refinements to algorithm functionality discussed in [Weihe 1997].) This mixin could have the form:[2]

```
template <class Graph> class Counting: public Graph {
  int nodes_visited, edges_visited;
public:
  Counting() : Graph() { nodes_visited = edges_visited = 0; }

  node succ_node (node v) {
    nodes_visited++;
    return Graph::succ_node(v);
  }

  edge succ_edge (edge e) {
    edges_visited++;
    return Graph::succ_edge(e);
  }
```

[2] We use C++ syntax for most of the examples of this section, in the belief that concrete syntax clarifies, rather than obscures, our ideas. To facilitate readers with limited C++ expertise, we avoid several cryptic idioms or shorthands (for instance, constructor initializer lists are replaced by assignments, we do not use the "struct" keyword to declare classes, etc.).

```
  // example method that displays the cost of an algorithm in
  // terms of nodes visited and edges traversed
  void report_cost () {
    cout << "The algorithm visited " << nodes_visited <<
            " nodes and traversed " << edges_visited <<
            " edges\n";
  }
  ... // other methods using this information may exist
};
```

By expressing operation counting as a mixin we ensure that it is applicable to many classes that have the same interface (i.e., many different kinds of graphs). Clearly, the implicit assumption is that classes, like `Dgraph` and `Ugraph`, have been designed so that they export similar interfaces. By standardizing certain aspects of the design, like the method interfaces for different kinds of graphs, we gain the ability to create mixin classes that can be reused in different occasions.[3] We can, for instance, use two different compositions:

```
typedef Counting < Ugraph > CountedUgraph;
```

and

```
typedef Counting < Dgraph > CountedDgraph;
```

to define a counted undirected graph type and a counted directed graph type. (We omit parameters to the graph classes for simplicity.) Note that the behavior of the composition is exactly what one would expect: any methods not affecting the counting process are exported (inherited from the graph classes). The methods that do need to increase the counts are "wrapped" in the mixin.

3.1.2 *Mixin Layers.* To implement entire collaborations as components we need to use mixins that encapsulate other mixins. We call the encapsulated mixin classes *inner mixins*, and the mixin that encapsulates them the *outer mixin.* Inner mixins can be inherited, just like any member variables or methods of a class. An outer mixin is called a *mixin layer* when *the parameter (superclass) of the outer mixin encapsulates all parameters (superclasses) of inner mixins.*[4] This is illustrated in Figure 3. `ThisMixinLayer` is a mixin that refines (through inheritance) `SuperMixinLayer`. `SuperMixinLayer` encapsulates three classes: `FirstClass`, `SecondClass`, and `ThirdClass`. `ThisMixinLayer` also encapsulates three inner classes. Two of them are mixins that refine the corresponding classes of `SuperMixinLayer`, while the third is an entirely new class.

Note that inheritance works at two different levels. First, a layer can inherit inner classes from the layer above it (for instance, `ThirdClass` in Figure 3). Second,

[3] Stated another way, a mixin defines a refinement of a class, but this refinement is not meaningful for every possible class. Standardized interfaces is a way to type or restrict the set of classes that a mixin can meaningfully refine. C++ syntax, in this regard, is unsatisfactory because C++ templates have untyped parameters. Languages like Pizza [Odersky and Wadler 1997] or GJ [Bracha et al. 1998] offer a better mechanism, where class parameters are typed by the interfaces that they implement. Unfortunately, Pizza and GJ do not support parameterized inheritance.

[4] Inner mixins can themselves be mixin layers.

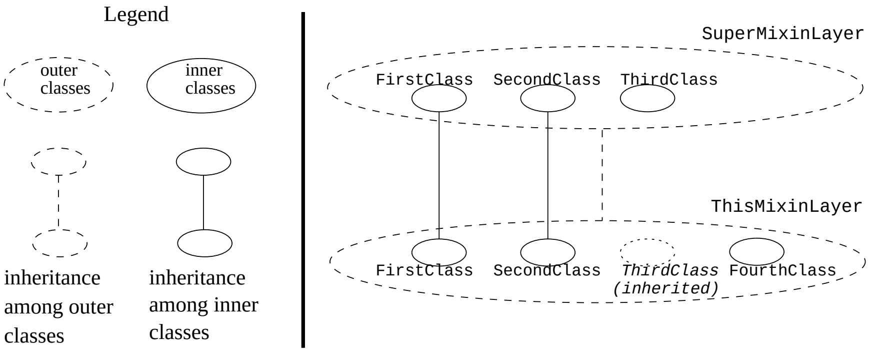


Fig. 3. Mixin layers schematically.

the inner mixins inherit member variables, methods, or other classes from their superclass.

3.1.3 *Mixin Layers in Various OO Languages.* The mixin layer concept is quite general and is not tied to any particular language idiom. Many flavors of the concept, however, can be expressed via specific programming language idioms: as stand-alone language constructs, as a combination of C++ nested classes and parameterized inheritance, as a combination of CLOS class-metaobjects and mixins, etc. We examine some of these different realizations next. The introduction of technical detail is necessary at this point as it helps us demonstrate concretely, in Section 3.2 , the advantages of mixin layers for implementing collaboration-based designs.

C++. We would like to support mixin layers in C++ using the same language mechanisms as those used for mixin classes. To do this, we can standardize the names used for inner class implementations (make them the same for all layers). This yields an elegant form of mixin layers that can be expressed using common C++ features. For instance, using C++ parameterized inheritance and nested classes, we can express `ThisMixinLayer` as a mixin layer (see again Figure 3) with two inner mixins (`FirstClass` and `SecondClass`) and one additional class (`FourthClass`):

```
template <class LayerSuper>
class ThisMixinLayer: public LayerSuper {
public:
  class FirstClass  : public LayerSuper::FirstClass  { ... };
  class SecondClass : public LayerSuper::SecondClass { ... };
  class FourthClass                                   { ... };
  ...
};
```

The above code fragment represents the form of mixin layers that we use in the examples of this section. Note that specifying a parameter for the outermost mixin automatically determines the parameters of all inner mixins. Composing mixin layers to form concrete classes is now as simple as composing mixin classes. If we have four mixin layers (`Layer1`, `Layer2`, `Layer3`, `Layer4`), we can compose them as:

```
 Layer4 < Layer3 < Layer2 < Layer1 > > >
```

where "`<...>`" is the C++ operator for template instantiation. Note that `Layer1` has to be a concrete class (i.e., not a mixin class). Alternatively we can have a class with empty inner classes that is the root of all compositions. (A third alternative is to use a *fixpoint* construction and instantiate the topmost layer with the result of the entire composition. This pattern has several desirable properties and is analyzed further in Chapter 3 of [Smaragdakis 1999].)

In the above code fragment we mapped the main elements of the mixin layer definition to specific implementation techniques. We used nested classes to implement class encapsulation. We also used parameterized inheritance to implement mixins. However, there are very different ways of encoding the same concept in other languages.

CLOS (and other reflective languages). We can encode mixin layers in CLOS [Kiczales et al. 1991] (and other reflective systems) by simulating their main elements using reflection (classes as first-class entities). Due to lack of space, we elide the implementation specifics. A discussion can be found in [Smaragdakis and Batory 1998] and [Smaragdakis 1999]. CLOS mixin layers are not semantically equivalent to C++ mixin layers (for instance, there is no default class data hiding: class members are by default accessible from other code in CLOS). Nevertheless, the two versions of mixin layers are just different flavors of the same idea.

Our ideas are applicable to other reflective languages. Smalltalk, in particular, has been a traditional test-bed for mixins, both for researchers (e.g., [Bracha and Griswold 1996; Mezini 1997; Steyaert et al. 1993]) and for practitioners [Montlick 1996]. A straightforward (but awkward) way to implement mixins in Smalltalk is as *class-functors*; that is, mixins can be functions that take a superclass as a parameter and return a new subclass.

Java. The Java language is an obvious next candidate for mixin layers. Java has no support for mixins and it is unlikely that the core language will include mixins in the near future. As will be described in Section 4, we extended the Java language with constructs that capture mixins and mixin layers explicitly. In this effort we used our JTS set of tools [Batory et al. 1998] for creating compilers for domain-specific languages. The system supports mixins and mixin layers through parameterized inheritance and class nesting, in much the same way as in C++.[5] Additionally, the fundamental building blocks of JTS itself were expressed as mixin layers, resulting in an elegant bootstrapped implementation. More on JTS in Section 4 .

Adding mixins to Java is also the topic of other active research [Agesen et al. 1997; Flatt et al. 1998] (although such work is almost certain to remain in the research domain). The work of [Flatt et al. 1998] presented a semantics for mixins in Java. This is particularly interesting from a theoretical standpoint as it addresses issues

[5] The Java 1.1 additions to the language [Sun Microsystems 1997] support nested classes and interfaces (actually both "nested" classes as in C++ and *member* classes—where nesting has access control implications). Nested classes can be inherited just like any other members of a class.

of mixin integration in a type-safe framework. As we saw, mixins can be expressed in C++ using parameterized inheritance. There have been several recent proposals for adding parameterization/genericity to Java [Agesen et al. 1997; Odersky and Wadler 1997; Bracha et al. 1998; Myers et al. 1997; Thorup 1997], but only the first [Agesen et al. 1997] supports parameterized inheritance and, hence, can express mixin layers.

It is interesting to examine the technical issues involved in supporting mixins in Java genericity mechanisms. Three of these mechanisms [Odersky and Wadler 1997; Bracha et al. 1998; Thorup 1997] are based on a *homogeneous* model of transformation: the same code is used for different instantiations of generics. This is not applicable in the case of parameterized inheritance—different instantiations of mixins are not subclasses of the same class (see [Agesen et al. 1997] for more details). Additionally, there may be conceptual difficulties in adding parameterized inheritance capabilities: the genericity approach of [Thorup 1997] is based on virtual types. Parameterized inheritance can be approximated with virtual types by employing *virtual superclasses* [Madsen and Møller-Pedersen 1989], but this is not part of the design of [Thorup 1997].

The approaches of Myers et al. [1997] and Agesen et al. [1997] are conceptually similar from a language design standpoint. Even though parameterized implementations do not directly correspond to types in the language (in the terminology of [Cardelli and Wegner 1985] they correspond to *type operators*), parameters can be explicitly constrained. This approach, combined with a *heterogeneous* model of transformation (i.e., one where different instantiations of generics yield separate entities) is easily amenable to adding parameterized inheritance capabilities, as was demonstrated in [Agesen et al. 1997].

3.2 Implementing Collaboration-Based Designs

Given the mixin layer concept, we can now express collaboration-based designs directly at the implementation level. We show how mixin layers can be used to perform the task and examine how it compares to two previous approaches. One is the straightforward implementation technique of application frameworks [Johnson and Foote 1988] using just objects and inheritance. The other is the technique of VanHilst and Notkin that employs C++ mixins to express individual roles.

3.2.1 *Using Mixin Layers.* A collaboration can be expressed by a mixin layer. The roles played by different objects are expressed as nested classes inside the mixin layer. The general pattern is:

```
template <class CollabSuper>
class CollabThis : public CollabSuper {
public:
  class FirstRole  : public CollabSuper::FirstRole  { ... };
  class SecondRole : public CollabSuper::SecondRole { ... };
  class ThirdRole  : public CollabSuper::ThirdRole  { ... };
  ...         // more roles
};
```

Again, mixin layers are composed by instantiating a layer with another as its parameter. This produces two classes that are linked as a parent-child pair in the inheritance hierarchy. For four mixin layers, `Collab1`, `Collab2`, `Collab3`, `FinalCollab` of the above form, we can define a class `T` that expresses the final product of the composition as:

```
typedef Collab1 < Collab2 < Collab3 < FinalCollab > > >  T ;
```

or (alternatively):

```
class T : public Collab1 < Collab2 < Collab3 < FinalCollab > > >
{ /* empty body */ };
```

In this paper, we consider these two forms to be equivalent.[6]

The individual classes that the original design describes are members (nested classes) of the above components. Thus, `T::FirstRole` defines the application class `FirstRole`, etc. Note that classes that do not participate in a certain collaboration can be inherited from collaborations above (we subsequently use the term "collaboration" for the mixin layer representing a collaboration when no confusion can result). Thus, class `T::FirstRole` is defined even if `Collab1` (the bottom-most mixin layer in the inheritance hierarchy) prescribes no role for it.

Example. Consider the graph traversal application of Section 2.2. Each collaboration is represented as a mixin layer. *Vertex Numbering*, for example, prescribes roles for objects of two different classes: *Vertex* and *Workspace*. Its implementation has the form:

```
template <class CollabSuper>
class NUMBER : public CollabSuper {
public:
  class Workspace : public CollabSuper::Workspace {
  ... // Workspace role methods
  };

  class Vertex : public CollabSuper::Vertex {
  ... // Vertex role methods
  };
};
```

Note how the actual application classes are nested inside the mixin layer. For instance, the roles for the *Vertex* and *Workspace* classes of Figure 1 correspond to `NUMBER::Vertex` and `NUMBER::Workspace`, respectively. Since roles are encapsulated, there is no possibility of name conflict. Moreover, we rely on the standardization of role names. In this example the names `Workspace`, `Vertex`, and `Graph` are used for roles in all collaborations. Note how this is used in the above code fragment: Any class generated by this template defines roles that inherit from classes `Workspace` and `Vertex` in its superclass (`CollabSuper`).

[6] There are differences, but these are a consequence of C++ policies and are not important for our discussion (they are discussed together with other C++ specific issues in [Smaragdakis 1999], Chapter 3).

Other collaborations of our Section 2.2 design are similarly represented as mixin layers. Thus, we have a DFT and a UGRAPH component that capture the *Depth-First Traversal* and *Undirected Graph* collaborations respectively. For instance, methods in the Vertex class of the DFT mixin layer include visitDepthFirst and isVisited (with implementations as suggested by their names). Similarly, methods in the Vertex class of UGRAPH include addNeighbor, firstNeighbor, and nextNeighbor, essentially implementing a graph as an adjacency list.

To implement default work methods for the depth-first traversal, we use an extra mixin layer, called DEFAULTW. The DEFAULTW mixin layer provides the methods for the Graph and Vertex classes that can be overridden by any graph algorithm (e.g., *Vertex Numbering*) used in a composition.

```
template <class CollabSuper>
class DEFAULTW : public CollabSuper {
public:
  class Vertex : public CollabSuper::Vertex {
  protected:
    bool workIsDone( CollabSuper::Workspace* )          {return 0;}
    void preWork( CollabSuper::Workspace* )             {}
    void postWork( CollabSuper::Workspace* )            {}
    void edgeWork( Vertex*, CollabSuper::Workspace* )   {}
  };

  class Graph : public CollabSuper::Graph {
  protected:
    void regionWork( Vertex*, CollabSuper::Workspace* ) {}
    void initWork( CollabSuper::Workspace* )            {}
    bool finishWork( CollabSuper::Workspace* )          {return 0;}
  };
};
```

The introduction of DEFAULTW (as a component separate from DFT) is an implementation detail, borrowed from the VanHilst and Notkin implementation of this example [1996b]. Its purpose is to avoid dynamic binding and enable multiple algorithms to be composed as separate refinements of more than one DFT component. This topic is discussed in detail during the comparison of mixin layers and application frameworks (Section 3.2.2).

With the collaboration entities of the original design represented as distinct mixin layers, it is easy to produce an entire application by composing collaborations. In fact, the mixin layers defined can be used to implement a *product-line*: a family of related applications. Different compositions of layers yield different products (members) of the family. In our example, the building blocks are the undirected graph, depth first traversal, etc. collaborations. We show the collaborations that are composed to build the vertex numbering graph application in Figure 4. We will soon explain what this composition means but first let us see how the different classes are related. The final implementation classes are members of the product of the composition, NumberC (e.g., NumberC::Graph is the concrete graph class). Figure 5 shows the mixin layers and their member classes, which represent roles, as

```
typedef DFT < NUMBER < DEFAULTW < UGRAPH > > > NumberC;
```

Fig. 4. A composition implementing the vertex numbering operation

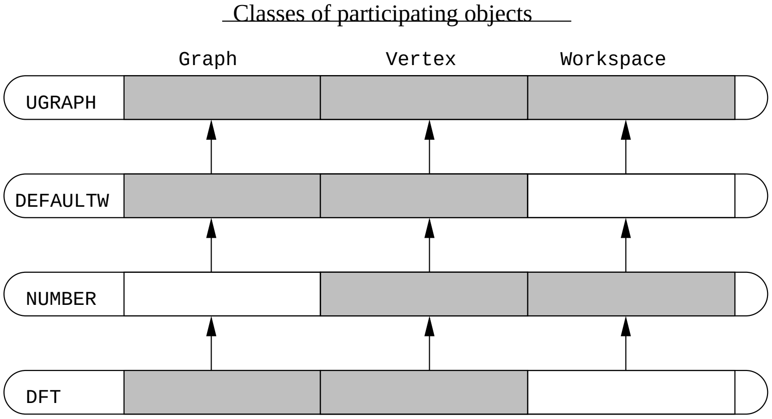


Fig. 5. Mixin-layers (ovals) and role-members (rectangles inside ovals) in the composition. Every component inherits from the one above it. Shaded role-members are those contained in the collaboration, unshaded are inherited. Arrows show inheritance relationships drawn from subclass to superclass.

they are actually composed. Each component inherits from the one above it. That is, `DFT` inherits role-members from `NUMBER`, which inherits from `DEFAULTW`, which inherits from `UGRAPH`. At the same time, `DFT::Graph` inherits methods and variables from `NUMBER::Graph`, which inherits from `DEFAULTW::Graph`, which inherits from `UGRAPH::Graph`. This double level of inheritance is what makes the mixin-layer approach so powerful. For instance, even though `NUMBER` does not specify a `Graph` member, it inherits one from `DEFAULTW`. The simplicity that this design affords becomes apparent in the following sections, when we compare it with alternatives.

The interpretation of the composition in Figure 4 is straightforward. It expresses the development of a vertex numbering application as a series of refinements. One begins with the `UGRAPH` mixin layer that implements an undirected graph. Next, default classes and methods that are common to all graph traversal algorithms are added by the mixin layer `DEFAULTW`. Then the algorithms and data members that are specific for vertex numbering are introduced by the `NUMBER` mixin layer. These algorithms, by themselves, are insufficient for performing vertex numbering because they rely on graph search algorithms which have yet to be added. Finally, the graph search algorithms—in this case, depth first search—are grafted on by the `DFT` mixin layer thereby completing the specification and implementation of this application.

Thus, every mixin layer except `UGRAPH` is implemented in terms of the ones above it. For instance, `DFT` is implemented in terms of methods supplied by `NUMBER`, `DEFAULTW`, and `UGRAPH`. An actual code fragment from the `visitDepthFirst` method implementation in `DFT::Vertex` is the following:

```
for ( v = (Vertex*)firstNeighbor(); v != NULL;
      v = (Vertex*)nextNeighbor() )
{
  edgeWork(v, workspace);
  v->visitDepthFirst(workspace);
}
```

The firstNeighbor, nextNeighbor, and edgeWork methods are not implemented by the DFT component. Instead they are inherited from components above it in the composition. firstNeighbor and nextNeighbor are implemented in the UGRAPH component (as they encode the iteration over nodes of a graph). edgeWork is a traversal refinement and (in this case) is implemented by the NUMBER component.

We can now see how mixin layers are both reusable and interchangeable. The DFT component of Figures 4 and 5 is oblivious to the *implementations* of methods in components above it. Instead, DFT only knows the *interface* of the methods it expects from its parent. Thus, the code above represents a skeleton expressed in terms of abstract operations firstNeighbor, nextNeighbor, and edgeWork. Changing the implementation of these operations merely requires the swapping of mixin layers. For instance, we can create an application (CycleC) that checks for cycles in a graph by replacing the NUMBER component with CYCLE:

```
typedef DFT < CYCLE < DEFAULTW < UGRAPH > > > CycleC;
```

The results of compositions (CycleC above and NumberC in Figure 4) can be used by a client program as follows: First, an instance of the nested Graph class (NumberC::Graph or CycleC::Graph) needs to be created. Then, Vertex objects are added and connected in the graph (the Graph role in mixin-layer UGRAPH defines methods addVertex and addEdge for this purpose). After the creation of the graph is complete, calling method depthFirst on it executes the appropriate graph algorithm.

Mixin layers are the building blocks of a graph application product-line. Each mixin layer is a reusable component and different members (i.e., products) of the family can be created by using different compositions of mixin layers. Note that no direct editing of the component is necessary and multiple copies of the same component can co-exist in the same composition. For instance, we could combine two graph algorithms by using two instances of the DFT mixin layer (in the same inheritance hierarchy), refined to perform a different operation each time:

```
class NumberC : public DFT < NUMBER < DEFAULTW < UGRAPH > > > {};
class CycleC  : public DFT < CYCLE < NumberC > > {};
```

Both algorithms can be invoked, depending on whether we access the depth-first traversal through a NumberC or a CycleC reference:

```
CycleC::Graph *graph_c = new CycleC::Graph();
NumberC::Graph *graph_n = graph_c;
```

Now a call to graph_c->depth_first invokes the cycle checking algorithm, while a call to graph_n->depth_first calls the vertex numbering algorithm. (Alternatively, we can qualify method names directly, e.g., graph_c->NumberC::Graph::depth_first(...) .)

As another example, the design may change to accommodate a different underlying model. For instance, operations could now be performed on directed graphs. The corresponding update (DGRAPH replaces UGRAPH) to the composition is straightforward (assuming that the algorithms are still valid for directed graphs as is the case with Holland's original implementation of this example [1992]):

```
typedef DFT < NUMBER < DEFAULTW < DGRAPH > > > NumberC;
```

Again, note that the interchangeability property is a result of the independence of collaborations.[7] A single UGRAPH collaboration completely incorporates all parts of an application that relate to maintaining an undirected graph (although these parts span several different classes). The collaboration communicates with the rest of the application through a well-defined and usually narrow interface.

For this and other similar examples, the reusability and interchangeability of mixin layers solves the *library scalability problem* [Batory et al. 1993; Biggerstaff 1994]: there are n features and often more than $n!$ valid combinations (because composition order matters and feature replication is possible [Batory and O'Malley 1992]). Hard-coding all different combinations leads to libraries of exponential size: the addition of a single feature can double the size of a library. Instead, we would like to have a collection of building blocks and compose them appropriately to derive the desired combination. In this way, the size of the library grows linearly in the number of features it can express (instead of exponentially, or super-exponentially).

Multiple Collaborations in a Single Design. An interesting question is whether mixin layers can be used to express collaboration-based designs where a single collaboration is instantiated more than once with the same class playing different roles in each instantiation. The answer is positive, and the desired result can be effected using *adaptor* mixin layers. Adaptor layers add no implementation but adapt a class so that it can play a pre-defined role. That is, adaptor layers contain classes with empty bodies that are used to "redirect" the inheritance chain so that predefined classes can play the required roles.

Consider the case of a producer-consumer collaboration, which was briefly discussed in Section 2.1. Our example is from the domain of compilers. A parser in a compiler can be viewed as a consumer of tokens produced by a lexical analyzer. At the same time, however, a parser is a producer of abstract syntax trees (consumed, for instance, by an optimizer). We can reuse the same producer-consumer collaboration to express both of these relationships. The reason for wanting to provide a reusable implementation of the producer-consumer functionality is that it could be quite complex. For instance, the buffer for produced-consumed items may be guarded by a semaphore, multiple consumers could exist, etc. The mixin layer implementing this collaboration takes Item as a parameter, describing the type of elements produced or consumed:

[7] By "independence" we mean that collaborations are composable because they conform to a particular design—all collaborations use Graph, Vertex, and Workspace classes with standardized methods. Given this standardization, the interchangeability—or independence—of these collaborations is achieved.

```
template <class CollabSuper, class Item>
class PRODCONS : public CollabSuper {
public:
  class Producer : public CollabSuper::Producer {
    void produce(Item item) { ... }
    // The functionality of producing Items is defined here
    ... // other Producer role methods
  };

  class Consumer : public CollabSuper::Consumer {
    Item consume() { ... }
    // The functionality of consuming Items is defined here
    ... // other Consumer role methods
  };
};
```

That is, PRODCONS adds the generic “produce” functionality to the Producer class and adds generic “consumer” functionality to the Consumer class.

Now we can use two simple adaptors to make a single class (Parser) be both a producer and a consumer in two different collaborations. The first adaptor (PRODADAPT) expresses the facts that a producer is also going to be a consumer (the actual consumer functionality is to be added later) and that the Optimizer class inherits the existing consumer functionality. This adaptor is shown below:

```
template <class CollabSuper>
class PRODADAPT : public CollabSuper {
public:
  class Consumer  : public CollabSuper::Producer {};
  class Optimizer : public CollabSuper::Consumer {};
  class Producer                                 {};
};
```

The second adaptor (CONSADAPT) is similar:

```
template <class CollabSuper>
class CONSADAPT : public CollabSuper {
public:
  class Lexer  : public CollabSuper::Producer    {};
  class Parser : public CollabSuper::Consumer    {};
};
```

Now a single composition can contain two copies of the PRODCONS mixin layer, appropriately adapted. For instance:

```
typedef COMPILER < CONSADAPT < PRODCONS <
                   PRODADAPT < PRODCONS < ..., Tree> >, Token > > >
  CompilerApp;
```

In the above, the COMPILER mixin layer is assumed to contain the functionality of a compiler that defines three classes, Lexer, Parser, and Optimizer. These

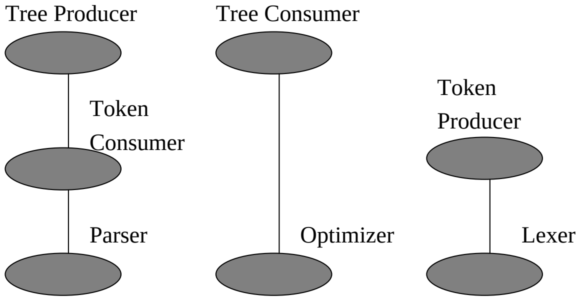


Fig. 6. The desired inheritance hierarchy has a `Parser` inheriting functionality both from a consumer class (a `Parser` is a consumer of tokens) and a producer class (a `Parser` is a producer of trees).

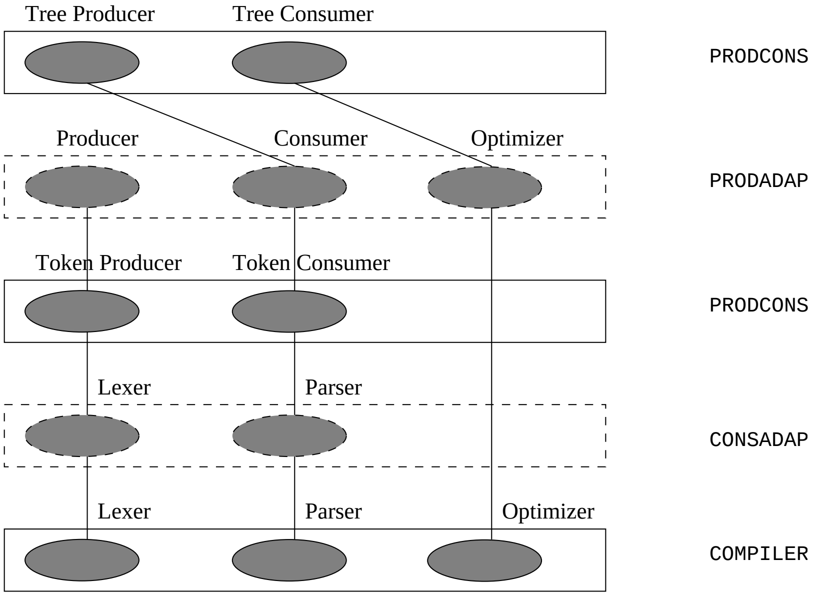


Fig. 7. By using adaptor layers (dotted rectangles), one can emulate the inheritance hierarchy of Figure 6, using only pre-defined mixin layers (solid rectangles). Since a single mixin layer (`PRODCONS`) is instantiated twice, adaptors help determine which class will play which role every time.

classes use the functionality supplied by the producer-consumer mixin layer. For instance, there may be a `parse` method in `COMPILER::Parser` that repeatedly calls the `consume` and `produce` methods. To better illustrate the role of adaptors, Figures 6 and 7 show the desired inheritance hierarchy for this example, as well as the way that adaptors are used to enable emulating this hierarchy using only predefined mixin layers. Note that each of the layers participating in the above composition appears as a rectangle in Figure 7.

3.2.2 *Comparison to Application Frameworks.*

In object-oriented programming, an *abstract* class cannot be instantiated (i.e., it cannot be used to create objects) but is only used to capture the commonalities of other classes. These classes inherit the common interface and functionality of the abstract class. An *object-oriented application framework* (or just *framework*) consists of a suite of interrelated abstract classes that embodies an abstract design for software in a family of related systems [Johnson and Foote 1988]. Each major component of the system is represented by an abstract class. These classes contain dynamically bound methods (`virtual` in C++), so that the framework user can add functionality by creating subclasses and overriding the appropriate methods. Thus, frameworks have the advantage of allowing reuse at a granularity larger than a single abstract class. But frameworks have the disadvantage that using them means manually making the client classes inherit from framework classes. Thus, the framework classes cannot easily be interchanged (with a different, similar framework) and the client classes cannot be reused in a different context—they are hard-wired to the framework.

In a *white-box framework*, users specify system-specific functionality by adding *methods* to the framework's classes. Each method must adhere to the *internal* conventions of the classes. Thus, using white-box frameworks is difficult, because it requires knowledge of their implementation details. In a *black-box framework*, the system-specific functionality is provided by a set of classes. These classes must adhere only to the proper *external* interface. Thus, using black-box frameworks is easier, because it does not require knowledge of their implementation details. Using black-box frameworks is further simplified when they include a library of pre-written functionality that can be used as-is with the framework.

Frameworks can be used to implement collaboration-based designs, but the amount of flexibility and modularity they can afford is far from optimal. The reason is that frameworks allow the reuse of abstract classes but have no way of specifying collections of concrete classes that can be used at will (i.e., either included or not and in any order) to build an application ([Batory et al. 2000]). Intuitively, frameworks allow reusing the skeleton of an implementation but not the individual pieces that are built from the skeleton. This can be seen through a simple combinatorics argument. Consider a set of four features, A, B, C, and D that can be combined arbitrarily to yield complete applications. For simplicity, assume that feature A is always first, and that no feature repetition is allowed. Then a framework may encode feature combination AB, thus allowing the user to program combinations $ABCD$ and $ABDC$. Nevertheless, these combinations must be coded separately (i.e., they cannot use any common code other than their common prefix, AB). The reason is that each instantiation of the framework creates a separate inheritance hierarchy and reusing a combination is possible only if one can inherit from one of its (intermediate or final) classes—only common prefixes are reusable. In our four-feature example, combinations that have no common prefix with the framework (for instance, ACD) simply cannot take advantage of it and have to be coded separately. This amounts to exponential redundancy for complex domains.

In the general case, assume a simple cost model that assigns one cost unit to each re-implementation of a feature. If feature order matters but no repetitions are possible, the cost of implementing all possible combinations using frameworks is equal to the number of combinations (each combination of length k differs by one feature from its prefix of length $k-1$). Thus, for n features, the total cost for implementing all combinations using frameworks is $\sum_{k=1}^{n} \frac{n!}{(n-k)!}$. (This number is derived by considering the sum of the feature combinations of length k, for each k from 1 to n.) In contrast, the cost of using mixin layers for the same implementation is equal to n—each component is implemented once and can be combined in arbitrarily many ways. With mixin layers, even compositions with no common prefixes share component implementations.

Even though our combinatorics argument represents an extreme case, it is reflective of the inflexibility of frameworks. For instance, optional features are common in practice and frameworks cannot accommodate them, unless all combinations are explicitly coded by the user. This is true even for domains where feature composition order does not matter or features have a specific order in which they must be used.

Another disadvantage of using frameworks to implement collaboration-based designs comes from the use of dynamically bound methods in frameworks. Even though the dynamic dispatch cost is sometimes negligible or can be optimized away, often it imposes a run-time overhead, especially for fine-grained classes and methods. With mixin layers, this overhead is avoided, as there is little need for dynamic dispatch. The reason is that mixin layers can be ordered in a composition so that most of the method calls are to their parent layers.

This reveals a general and important difference between mixin-based programming and standard object-oriented programming. When a code fragment in a conventional OO class needs to be generic, it is implemented in terms of dynamically bound methods. These methods are later overridden in a subclass of the original class, thus refining it for a specific purpose. With mixin classes, the situation is different. A method in a mixin class can define generic functionality by calling methods in the class's (yet undefined) *superclass*. That is, generic calls for mixins can be both up-calls and down-calls in the inheritance hierarchy. Generic up-calls are specialized statically, when the mixin class's superclass is set. Generic down-calls provide the standard OO run-time binding capabilities. Their use can be limited to cases where the exact version of the method to be called is truly not known until run-time. In contrast, in application frameworks, dynamic binding is often used just for modularity reasons (calling functionality without yet having defined it) even if the target ends up being known statically. This can be eliminated in a mixin-based approach because we are allowed to add functionality to a mixin class's superclass. Refinement of existing functionality is not just a top-down process but involves composing mixins arbitrarily, often with many different orders being meaningful.

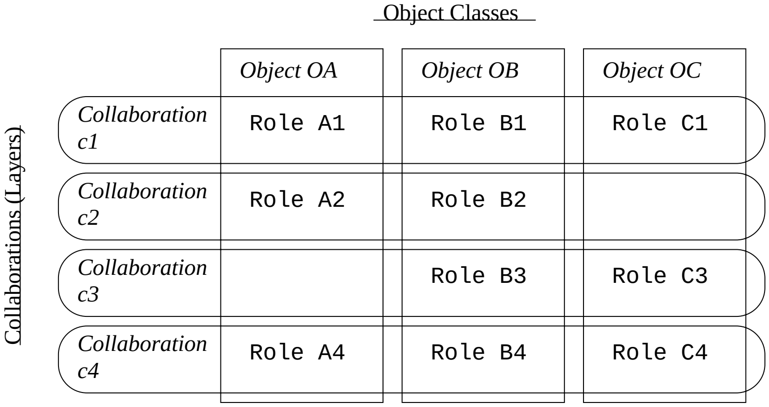
Example. We illustrate the above points with the graph algorithm example of Section 2.2. The original implementation of this application [Holland 1992] used a black-box application framework on which the three graph algorithms were implemented. The framework consists of the implementations of the `Graph`, `Vertex`, and `Workspace` classes for the *Undirected Graph* and *Depth First Traversal* collaborations. The classes implementing the depth-first traversal have methods like preWork, postWork, edgeWork, etc., *which are declared to be dynamically bound* (virtual in C++). In this way, any classes inheriting from the framework classes can refine the traversal functionality by redefining the operation to be performed the first time a node is visited, when an edge is traversed, etc.

VanHilst and Notkin discussed the framework implementation of this example in detail [1996b]. Our presentation here merely adapts their observations to our above discussion of using frameworks to implement collaboration-based designs. A first observation is that, in the framework implementation, the base classes are fixed and changing them requires hand-editing (usually copying and editing, which results in redundant code). For instance, consider applying the same algorithms to a directed, as opposed to an undirected graph. If both combinations need to be used in the same application, code replication is necessary. The reason is that the classes implementing the graph algorithms (e.g., *Vertex Numbering*) must have a fixed superclass. Hence, two different sets of classes must be introduced, both implementing the same graph algorithm functionality but having different superclasses.

A second important observation pertains to our earlier discussion of optional features in an application. In particular, a framework implementation does not allow more than one refinement to co-exist in the same inheritance hierarchy. Thus, unlike the mixin layer version of the code in Section 3.2.1, with frameworks we cannot have a single graph that implements both the *Vertex Numbering* and the *Cycle Checking* operations. The reason is that the dynamic binding of methods in the classes implementing the depth-first traversal causes the most refined version of a method to be executed on every invocation. Thus, multiple refinements cannot coexist in the same inheritance hierarchy since the bottom-most one in the inheritance chain always supersedes any others. In contrast, the flexibility of mixin layers allows us to break the depth-first traversal interface in two (the DEFAULTW and the DFT component, discussed earlier) so that DFT calls the refined methods *in its superclass* (i.e., without needing dynamic binding). In this way, multiple copies of the DFT component can co-exist and be refined separately. At the same time, obviating dynamic binding results into a more efficient implementation—dynamic dispatch incurs higher overhead than calling methods of known classes (although sometimes it can be optimized by an aggressive compiler).

3.2.3 *Comparison to the VanHilst and Notkin Method.* The VanHilst and Notkin approach [1996b; 1996c; 1996a; 1997] is another technique that can be used to map collaboration-based designs into programs. The method employs C++ mixin classes, which offer the same flexibility advantages over a framework implementation as the mixin layers approach. Nevertheless, the components represented by VanHilst and Notkin are small-scale, resulting in complicated specifications of their interdependencies.

VanHilst and Notkin use mixins in C++ to represent roles. More specifically, each individual role is mapped to a different mixin and is also parameterized by any other classes that interact with the given role in its collaboration. For an example, consider role B4 in Figure 8 (which replicates Figure 1 for easy reference). This role participates in a collaboration together with two other roles, A4 and C4. Hence, it needs to be aware of the classes playing the two roles (so that, for instance,



	Object OA	Object OB	Object OC
Collaboration c1	Role A1	Role B1	Role C1
Collaboration c2	Role A2	Role B2	
Collaboration c3		Role B3	Role C3
Collaboration c4	Role A4	Role B4	Role C4

Fig. 8. Example collaboration decomposition. Ovals represent collaborations, rectangles represent objects, their intersections represent roles.

it can call appropriate methods). With the VanHilst and Notkin technique, the role implementation would be a mixin that is parameterized by these two extra classes:

```
template <class RoleSuper, class OA, class OC>
class B4 : public RoleSuper {
   ... /* role implementation, using OA, OC */
};
```

Consider that the actual values for parameters `OA`, `OC` would themselves be the result of template instantiations, and their parameters also, and so on (up to a depth equal to the number of collaborations). This makes the VanHilst and Notkin method complicated even for relatively small examples. In the case of a composition of n collaborations, each with m roles, the VanHilst and Notkin method can yield a parameterization expression of length m^n. Additionally, the programmer has to explicitly keep track of the mapping between roles and classes, as well as the collaborations in which a class participates. For instance, the mixin for role `A4` in Figure 1 has to be parameterized with the mixin for role `A2`—the programmer cannot ignore the fact that collaboration *c3* does not specify a role for object *OA*. From a software evolution standpoint, local design changes cannot easily be isolated, since collaborations are not explicitly represented as components. These limitations make the approach unscalable: various metrics of programmer effort (e.g., length of composition expressions, parameter bindings that need to be maintained, etc.) grow exponentially in the number of features supported. (This is the same notion of scalability as in our earlier discussion of the library scalability problem.)

Conceptually, the scalability problems of the VanHilst and Notkin approach are due to the small granularity of the entities they represent: each mixin class represents a single role. Roles, however, have many external dependencies (for instance, they often depend on many other roles in the same collaboration). To avoid hard-coding such dependencies, we have to express them as extra parameters to the mixin class, as in the preceding code fragment. Reusable components should have

```
class NumberC: public DFT <NUMBER <DEFAULTW <UGRAPH> > > {};
class CycleC : public DFT < CYCLE < NumberC > >          {};
```

Fig. 9. Our mixin layer implementation of a multiple-collaboration composition. The individual classes are members of `NumberC`, `CycleC` (e.g., `NumberC::Vertex`, `CycleC::Graph`, etc.).

```
class Empty {};
class WS         : public WorkspaceNumber                {};
class WS2        : public WorkspaceCycle                 {};
class VGraph     : public VertexAdj<Empty>               {};
class VWork      : public VertexDefaultWork<WS,VGraph>   {};
class VNumber    : public VertexNumber<WS,VWork>         {};
class V          : public VertexDFT<WS,VNumber>          {};
class VWork2     : public VertexDefaultWork<WS2,V>       {};
class VCycle     : public VertexCycle<WS2,VWork2>        {};
class V2         : public VertexDFT<WS2,VCycle>          {};
class GGraph     : public GraphUndirected<V2>            {};
class GWork      : public GraphDefaultWork<V,WS,GGraph>  {};
class Graph      : public GraphDFT<V,WS,GWork>           {};
class GWork2     : public GraphDefaultWork<V2,WS2,Graph> {};
class GCycle     : public GraphCycle<WS2,GWork2>         {};
class Graph2     : public GraphDFT<V2,WS2,GCycle>        {};
```

Fig. 10. Same implementation using the VanHilst/Notkin approach. `V` corresponds to our `NumberC::Vertex`, `Graph` to `NumberC::Graph`, `WS` to `NumberC::Workspace`, etc.

few external dependencies, as made possible by using mixin layers to model collaborations.

Example. Consider a composition implementing both the *Cycle Checking* and the *Vertex Numbering* operation on the same graph. Recall that the ability to compose more than one refinement (or multiple copies of the same refinement) is an advantage of the mixin-based approach (both ours and the VanHilst and Notkin method) over frameworks implementations.

The components (mixins) used by VanHilst and Notkin are similar to the inner classes in our mixin layers, with extra parameters needed to express their dependencies with other roles in the same collaboration. Our specification is shown in Figure 9 (reproducing a previously presented code fragment). A compact representation of a VanHilst and Notkin specification is shown in Figure 10. (A more readable version of the same code included in [VanHilst and Notkin 1996b] is even lengthier.)[8]

Figure 10 makes apparent the complications of the VanHilst/Notkin approach. Each mixin representing a role can have an arbitrary number of parameters and can instantiate a parameter of other mixins. In this way, parameterization expressions of exponential (to the number of collaborations) length can result. To alleviate this problem, the programmer has to introduce explicitly intermediate types that encode common sub-expressions. For instance, `V` is an intermediate type in Figure 10. Its only purpose is to avoid introducing the sub-expression `VertexDFT<WS,VNumber>` three different times (wherever `V` is used). Of course,

[8] The object code of both is, as expected, of almost identical size.

VNumber itself is also just a shorthand for VertexNumber<WS,VWork>. VWork, in turn, stands for VertexDefaultWork<WS,VGraph>, and so on.[9] Additional complications arise when specifying a composition: users must know the number and position of each parameter of a role-component. Both of the above requirements significantly complicate the implementation and make it error-prone.

Using mixin layers, the exponential blowup of parameterization expressions is avoided. Every mixin layer only has a single parameter (the layer above it). By parameterizing a mixin layer A by B, A becomes implicitly parameterized by all the roles of B. Furthermore, if B does not contain a role for an object that A expects, it will inherit one from above it. This is the benefit of expressing the collaborations themselves as classes: they can extend their interface using inheritance.

Another practical advantage of mixin layers is that it encourages consistent naming for roles. Hence, instead of explicitly giving unique names to role-members, we have standard names and only distinguish instances by their enclosing mixin layer. In this way, VertexDFT, GraphDFT, and VertexNumber become DFT::Vertex, DFT::Graph and NUMBER::Vertex, respectively.

In [1996b], VanHilst and Notkin questioned the scalability of their method. One of their concerns was that the composition of large numbers of roles "can be confusing even in small examples..." The observations above (length of parameterization expressions, number of components, consistent naming) show that mixin layers address this problem and do scale gracefully, without losing the advantages of the VanHilst and Notkin implementation.

3.3 Mixin Layers Considerations

We have argued that mixin layers are better for implementing collaboration-based designs than other alternatives. Nevertheless, mixin layers are certainly not a "silver bullet". They are good for in-house development of product-line architectures for mature domains and require programming language and tool support for specification and debugging. These points are analyzed below in more detail, but we note that they are by no means specific to mixin layers: other competitive techniques (e.g., application frameworks, or the VanHilst and Notkin method) have similar restrictions.

—*Appropriate Domains for Mixin Layers*: Mixin layers are not appropriate for every domain. In general, the most suitable domains are mature, well-understood, amenable to detailed decompositions and elaborations of collaboration-based designs. The domain should be decomposable into largely independent refinements. Composing such refinements need not result in an increase in the level of abstraction. Instead, refinements can represent different concerns at the same conceptual level. (E.g., the addition of more operations on graphs does not alter the abstraction that we are still dealing with graphs; rather, adding more operations merely enriches the graph abstraction.) A well-known observation is that, even in strictly layered domains, like operating systems, the notion of "information module" does

[9] Some compilers (e.g., MS VC++, g++) internally expand template expressions, even though the user has explicitly introduced intermediate types. This caused page-long error messages for incorrect compositions when we experimented with the VanHilst and Notkin method, rendering debugging impossible.

not necessarily coincide with the notion of "layer of abstraction". Modules may encompass different parts of several layers [Habermann et al. 1976]. Mixin layers are a kind of "information module" and similar observations apply. Mixin layers lead to physically layered implementations, which may or may not have a negative impact on application performance. Mixin layers are implementations of a standard design imposed on a domain. In-house environments of individual companies are best to maintain this standard; open collaborative communities might make such standards difficult to follow. No precise quantification of these properties can be given, but a designer can usually assess the appropriateness of our techniques.

—*Difficulties in Using Mixin Layers*: Good OO designs limit the depth of inheritance hierarchies to a small number (e.g., 3). In contrast, compositions of mixin layers often leads to long inheritance chains. This can become problematic during debugging (chasing method calls up an inheritance hierarchy) and generally understanding where the functionality of a class is located on an inheritance chain. Another difficulty can be learning the order in which mixin layers can be composed. While this can be ameliorated by good tool support [Batory and Geraci 1997], it is something more that needs to be learned and composition rules need to be precisely stated.

—*Implementation Requirements for Mixin Layers and Interaction with Language Features*: Mixin layers are only as good as the technology to support them. Some of the proposed implementation techniques have specific technical disadvantages, especially in conjunction with particular compiler technology. For instance, our C++ template implementation of mixin layers may result in (binary) code duplication if the same layer is used multiple times in a composition. Nevertheless, no fundamental implementation drawbacks exist in relation to mixin layers. Implementation considerations for the C++ version of mixin layers are described in [Smaragdakis and Batory 2000].

Several general programming language issues arise in connection with mixin layers and their compositions. Most of these issues pertain to the interactions of mixin layers with type systems. Type information can be used to detect errors in a composition of mixin layers. At the same time, layers are defined in isolation and the problem of propagating type information between layers is especially interesting. Since the focus of this paper is not on concrete language solutions, we point the reader to [Smaragdakis 1999], Chapter 3, where such issues are analyzed in detail.

4. AN APPLICATION: THE JAKARTA TOOL SUITE

In this section, we discuss an application of mixin layers to a medium-size software project (about 30K lines of code). The project is the *Jakarta Tool Suite (JTS)* [Batory et al. 1998]—a set of language extensibility tools, aimed mainly at the Java language. We use mixin layers as the building blocks that form different versions of the *Jak* tool of JTS. Jak is the modular compiler in JTS. Different versions of Jak can be created using different combinations of layers. Layers may be responsible for type-checking, compiling, and/or creating code for a different set of language constructs. Additionally, layers may be used to add new functionality across a large group of existing classes. In this way, the user can design a language by putting together conceptual language "modules" (i.e., consistent sets of language constructs) and implement a compiler for this language as a version of Jak composed of the mixin layers corresponding to each language module. Currently available layers support the base Java language, meta-programming extensions, general purpose extensions (e.g., syntax macros for Java), a domain-specific language for data structure programming (P3), etc.

The choice of the compiler domain as a large-scale test case for mixin layers is not arbitrary. Compilers are well-understood, with modern compiler construction benefiting from years of formal development and stylized design patterns. The domain of compilers has been used several times in the past in order to demonstrate modularization mechanisms. Selectively, we mention the *visitor* design pattern [Gamma et al. 1995], which is commonly described using the example of a compiler with a class corresponding to each syntactic type that its parser can recognize (e.g., there is a class for if-statements, a class for declarations, etc.). In this case, the visitor pattern can be used to add new functionality to all classes, without distributing this functionality across the classes. Our application of mixin layers to the compilers domain has very much the same modularization flavor. We use mixin layers to isolate aspects of the compiler implementation, which can be added and removed at will. Compared to the visitor pattern, mixin layers offer greater capabilities—for instance, allowing the addition of state (i.e., member variables) to existing classes.

Overall, the outcome of applying mixin layers to JTS was very successful. The flexibility afforded by a layered design is essential in forming compilers for different language dialects. Additionally, mixin layers helped with the internal organization of the code, so that changes were easily localized. Additions that could be conceptually grouped together (like those reflecting the language changes from Java 1.0 to Java 1.1) were introduced as new mixin layers, without disrupting the existing design. JTS was thus easier to implement and has become easier to maintain.

We next discuss JTS and the use of mixin layers in its implementation. Section 4.1 offers some essential background in JTS by describing the way parsers are generated and initial class hierarchies are established based on language syntax. Section 4.2 discusses the actual application of mixin layers in JTS.

4.1 JTS Background: Bali as a Parser Generator

Bali is the JTS tool responsible for putting together compilers. Although Bali is a component-based tool, in this section we limit our attention to the more conventional grammar-specification aspects of Bali.

The syntax of a language is specified as a Bali grammar, which is an annotated BNF grammar extended with regular-expression repetitions. Bali transforms a Bali grammar into a lexical analyzer and parser. For example, two Bali productions are shown below: one defines `StatementList` as a sequence of one or more `Statements`, and the other defines `ArgumentList` as a sequence of one or more `Arguments` separated by commas.

```
StatementList : ( Statement )+ ;
ArgumentList : Argument ( ‘,’ Argument )*;
```

```
       // Lexeme definitions
"print"   PRINT
"+"       PLUS
"-"       MINUS
"("       LPAREN
")"       RPAREN
"[0-9]*"  INTEGER

%%        // production definitions
          // start rule is Action

Action  : PRINT Expr                      :: Print
        ;
Expr    : Expr PLUS Expr                  :: Plus
        | Expr MINUS Expr                 :: Minus
        | MINUS Expr                      :: UnaryMinus
        | LPAREN Expr RPAREN              :: Paren
        | INTEGER                         :: Integer
        ;
```

Fig. 11. A Bali Grammar for an Integer Calculator

Repetitions have been used before in the literature [Wirth 1977; Wile 1993; Reasoning Systems 1990]. They simplify grammar specifications and allow an efficient internal representation as a list of trees.

Bali productions are annotated by the class of objects that is to be instantiated when the production is recognized. For example, consider the Bali specification of the Jak `SelectStmt` rule:

```
SelectStmt
  : IF ‘(’ Expression ‘)’ Statement    ::IfStm
  | SWITCH ‘(’ Expression ‘)’ Block    ::SwStm
  ;
```

When a parser recognizes an “if” statement (i.e., an `IF` token, followed by ‘(‘, `Expression`, ‘)’, and `Statement`), an object of class `IfStm` is created. Similarly, when the pattern defining a “switch” statement (a `SWITCH` token followed by ‘(‘, `Expression`, ‘)’, and `Block`) is recognized, an object of class `SwStm` is created. As a program is parsed, the parser instantiates the classes that annotate productions, and links these objects together to produce the syntax tree of that program.

A Bali grammar specification is a streamlined document. It is a list of the lexical patterns that define the tokens of the grammar followed by a list of annotated productions that define the grammar itself. A Bali grammar for an elementary integer calculator is shown in Figure 11. From this grammar specification, Bali generates a lexical analyzer and a parser (we use the `JavaCC` lexer/parser generator as a backend).

Associating grammar rules with classes allows Bali to do more than generate a parser. In particular, Bali can deduce an inheritance hierarchy of classes representing different pieces of syntax. Consider Figure 12, which shows rules `Rule1` and `Rule2`. When an instance of `Rule1` is parsed, it may be an instance of `pattern1` (an object of class `C1`), or an instance of `Rule2` (an object of class `Rule2`). Similarly, an

```
Rule1 : pattern1   :: C1
      | Rule2
      ;

Rule2 : pattern2   :: C2
      | pattern3   :: C3
      ;
```

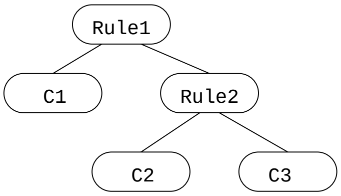


Fig. 12. Inferring inheritance hierarchies from grammar rules.

instance of `Rule2` is either an instance of `pattern2` (an object of `C2`) or an instance of `pattern3` (an object of `C3`). From this information, the inheritance hierarchy of Figure 12 is constructed: classes `C1` and `Rule2` are subclasses of `Rule1`, and `C2` and `C3` are subclasses of `Rule2`.

Additionally, for each production Bali infers the constructors for syntax tree node classes. Each parameter of a constructor corresponds to a token or nonterminal of a pattern.[10] For example, the constructor of the `IfStm` class has the following signature:

```
IfStm(Token iftk, Token lp, Expression exp, Token rp, Statement st)
```

Methods for editing and unparsing nodes are additionally generated.

Although Bali automatically generates an inheritance hierarchy and some methods for the produced Jak compiler, there are obviously many methods that cannot be generated automatically. These include type checking, reduction, and optimization methods. Such methods are syntax-type-specific; we hand-code these methods and encapsulate them as a mixin layer that contains subclasses of Bali-generated classes.

In essence, Bali takes the grammar specification and uses it to produce a skeleton for the compiler of the language. The skeleton has the form of a set of classes organized in an inheritance hierarchy, together with the methods that can be automatically produced (that is, constructors, editing, and unparsing methods). In other words, Bali produces an *application framework* for a compiler. The framework is encapsulated in a mixin layer that occupies the root of all mixin layer compositions implementing different versions of Jak.

4.2 Bali Components and Mixin Layers in JTS

Apart from its parser generator aspect, Bali is also a tool that synthesizes language implementations from components. Bali can create compilers for a family of languages, depending on the selection of components used as its input. This is essentially a product-line of language translators, with their common functionality factored out in reusable components. We use the name *Jak* for any Bali-generated compiler. Currently available Bali components support the base Java language,

[10] The tokens need not be saved. However, Bali-produced precompilers presently save all white space—including comments—with tokens. In this way, JTS-produced tools that transform domain-specific programs retain embedded comments. This is useful when debugging programs with a mixture of generated and hand-written code, and is a necessary feature if transformed programs are subsequently maintained by hand.

meta-programming extensions (e.g., code template operators), general purpose extensions (e.g., syntax macros for Java), a domain-specific language for state machines [Batory et al. 2000], and more. Compositions of these components define different variants of Jak (i.e., different members of a product-line of Java dialects): with/without meta-programming constructs, with/without state machine extensions, with/without data structure extensions, and so on. This is another instance of the library scalability problem [Batory et al. 1993; Biggerstaff 1994]. We want to compose the different variants of Jak from components encapsulating orthogonal units of functionality.

A *Bali component* has two parts: The first is a Bali grammar file (which contains the lexical tokens and grammar rules that define the syntax of the host language or language extension—for extensions that only change the semantics but not the syntax, this file is absent). The second is a mixin layer encapsulating a collection of multiple hand-coded classes that contain the reduction, type-checking, etc. methods for each syntax type defined in that grammar file.

To illustrate how classes are defined and refined in Bali, consider four concrete Bali components: `Java` is a component implementing the base Java language, `SST` implements code template operators like tree constructors and explicit escapes,[11] `GScope` supplies scoping support for program generation, and `P3` implements a language for data structures. The Jak language and compiler can be defined by a composition of these components. We use the `[...]` operator to designate component composition—for instance, `P3[GScope[SST[Java]]]`.

The syntax of a composed language is defined by taking the union of the sets of production rules in each Bali component grammar. The semantics of a composition is defined by composing the corresponding mixin layers. Figure 13 depicts the class hierarchy of the Jak compiler. `AstNode` belongs to the JTS kernel, and is the root of all inheritance hierarchies that Bali generates. Using the composition grammar file (the union of the grammar files for the `Java`, `SST`, `GScope`, and `P3` components), Bali generates a mixin layer that encapsulates the hierarchy of classes that contain tree node constructors, unparsing, and editing methods. Each remaining mixin layer then grafts onto this hierarchy its hand-coded classes. These define the reduction, optimization, and type-checking methods of tree nodes by refining existing classes. *The terminal classes of this hierarchy are those that are instantiated by the generated compiler.*

It is worth noting that Figure 13 is not drawn to scale. Jak consists of over 500 classes. The number of classes that a mixin layer adds to an existing hierarchy ranges from 5 to 40. Nevertheless, the simplicity and economy of specifying Jak using component compositions is enormous: to build the Jak compiler, all that users have to provide to Bali is the equation `Jak = P3[GScope[SST[Java]]]`, and Bali does the rest. To compose all these classes by hand (as would be required by Java) would be very slow, extremely tedious, and error prone. Additionally,

[11] Our code template operators are analogous to the backquote/unquote pair of Lisp operators. Unlike Lisp, however, multiple operators exist in JTS—one for each syntactic type (e.g., declaration, expression, etc.). Multiple constructors in syntactically rich languages are common (e.g., [Weise and Crew 1993], [Chiba 1996]). The main reason has to do with the ease of parsing code fragments.

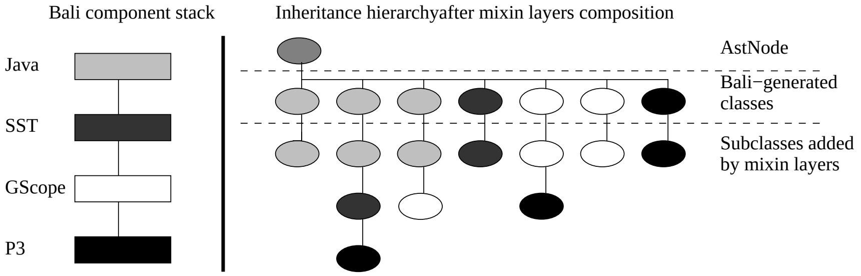

Bali component stack
Inheritance hierarchyafter mixin layers composition
Java
SST
GScope
P3
AstNode
Bali−generated classes
Subclasses added by mixin layers

Fig. 13. The Jak Inheritance Hierarchy.

the scalability advantages of mixin layers can easily be demonstrated: when new extension mechanisms or new base languages are specified as components, a subset of them can be selected and Bali automatically composes a compiler for the desired language variant.

4.3 Java Mixin Layers for JTS

In Section 3.1.3, we discussed the applicability of mixin layers in various programming languages. There we explained that Java already supports nested classes but the language currently specifies no parameterization mechanism. Furthermore, some of the proposed parameterization mechanisms for Java (e.g., Pizza [Odersky and Wadler 1997] or Thorup's virtual types [1997]) do not support parameterized inheritance. In order to support mixin layers for Bali components in JTS, we implemented our own Java language extensions for parameterization. This section gives a brief overview of the main language construct.

Our parameterization extensions to Java are geared towards mixin layer development (as opposed to general-purpose genericity). Our approach in designing and implementing these language constructs was motivated by pragmatic and not conceptual considerations: we needed a layer mechanism to facilitate our own development efforts—not to supply the best-designed and robust parameterization mechanism for Java. Therefore, our implementation was straightforward, adopting a heterogeneous model of transformation: for each instantiation of a mixin layer, a new Java class is created at the source code level. Thus, our approach resembles C++ template instantiation and does not take advantage of the facilities for load-time class adaptation offered by the Java Virtual Machine (see, e.g., the approach of Agesen et al. [1997] and the work on binary component adaptation [Keller and Hölzle 1998]). Nevertheless, in our context our approach is not necessarily at a disadvantage. Mixin layers in Bali component compositions are never reused in the same application (i.e., a single Jak compiler uses at most one instance of a mixin layer). Therefore, code bloat (redundancy in generated classes) is not a problem. At the same time, our straightforward approach made for an easier implementation which contributed to the faster development of JTS.

The implementation of our Java extensions for mixin layer support occurred concurrently with the development of JTS. In fact, an early version of JTS was used to implement the first version of our Java mixin layers. The Java mixin layers were, in turn, used to evolve and further develop JTS, resulting in a bootstrapped implementation. (Actually, this is not the only reason why JTS is based on a bootstrapped implementation. Another reason is that the meta-programming capabilities added to Java have been used in the code that implements JTS itself. The entire JTS system is compiled using a basic version of the Jak compiler, composed of only a few layers that specify the basic Java language, code template operators, syntax macros, etc.)

The syntax of mixin layers is straightforward and resembles their C++ counterparts. Two new keywords are introduced: `layer` and `realm`. The `layer` keyword is analogous to `class` but defines a mixin layer (i.e., an outer class that may be parameterized with respect to its superclass). The `realm` keyword is used to specify interface conformance for mixin layers, in analogy to the Java `implements` keyword. Finally, the `[...]` operator is used to specify layer composition. The (slightly simplified) general form of a layer definition is shown below, with the terminal symbols appearing in bold for clarity:

```
layer_definition :
  layer layer_name (param_list) realm realm_name [super]
  { declaration_list }
```

The syntax for non-terminals in the above definition is straightforward. `param_list` is a list of type parameters for the mixin layer. If the parameter list contains layers, the parameterization can be constrained by specifying the expected realm of these layers. The optional `super` construct designates an `extends` clause (in much the same way as for regular Java classes). The contents of a mixin layer can only be Java type declarations.

The actual details of our implementation are not important. We consider of much greater importance the general approach that this implementation represents. What we did in JTS is an example of a *domain-specific languages* approach to software construction. In the course of creating a medium-size software project, we recognized that mixin layers would facilitate our task significantly. That is, we saw an opportunity for improving our implementation through extra language support. It then proved cost-effective to add the extra linguistic constructs that were needed (i.e., mixin layers), in the course of implementing the original project (i.e., JTS).

It is our belief that the domain-specific language approach to software construction is a promising way to building better software. The designer of a software application can (and should) be thinking about language constructs that can have a significant impact in the application's efficiency, maintainability, or reusability. Often such constructs can be readily identified, but they are not available in the implementation language of choice. With the advent of language extensibility tools, as well as extensible/reflective programming languages, supplying special-purpose (or *domain-specific*) language support may be the right approach in fighting software complexity. JTS itself is a tool aiming at facilitating the implementation of domain-specific languages and language extensions. The use of mixin layers in the implementation of JTS is a vivid demonstration of the same paradigm that JTS promotes.

5. RELATED WORK

There is an enormous wealth of research in the area of component-based software construction and code modularization. Here we selectively discuss some approaches that are related to our work but have not been described previously in this paper.

5.1 GenVoca

GenVoca is a layered design and implementation methodology, mainly applied to application generators (i.e., compilers for domain-specific programming languages). GenVoca advocates that a domain be decomposed in terms of largely-orthogonal features which are implemented as layers. Applications in the domain can be synthesized by composing layers; layer composition is performed by a generator. The name "GenVoca" was derived from the first two generators that exhibited these principles: Genesis (extensible database systems) [Batory 1987; Batory et al. 1988] and Avoca (network protocols) [O'Malley and Peterson 1992]. GenVoca generators for other domains include: data manipulation languages [Villarreal 1994], distributed file systems [Heidemann and Popek 1994], host-at-sea buoy systems [Weiss 1990], and real-time avionics software [Coglianese and Szymanski 1993]. Mixin layers were originally inspired by the GenVoca model and are now an essential part of its arsenal of implementation techniques. Although we have not attempted full implementations, our experience suggests that mixin layers can be used to obtain many of the same benefits as full GenVoca generators for the above domains. That is, much of the benefit of GenVoca generators is due to the layering technology and not to the use of compiler techniques.

5.2 Modules in High-Level Languages

High-level languages often provide *modules* (a.k.a. *packages* or *namespaces*) as fundamental abstractions. Representative approaches include Ada *packages* [International Organization for Standardization 1995]—which is a prototypical modularization scheme for block structured languages, ML [Milner et al. 1990]—which provides a very powerful module system based on polymorphic types, Java *packages*, and C++ namespaces [Stroustrup 1997].[12]

Mixin layers are expressible in the latest incarnations of Ada (Ada95 [International Organization for Standardization 1995]). Standard ML still lacks support for extensible records (i.e., a counterpart of inheritance). Nevertheless, there is nothing fundamental that prevents integrating mixin layers. Recent research has brought some of the mixin layers ideas in a modular language framework. Findler and Flatt's work [1998] introduces constructs remarkably similar to mixin layers, in an experimental, module-based object system.

The most interesting lesson, however, is that modules—unlike classes—are often not well integrated in programming languages. For example, a C++ namespace

[12] It is perhaps debatable whether C++ namespaces and Java packages are modules, because they can be later re-opened and have more definitions added to them. Nevertheless, we choose to include these mechanisms here. In practice, they are often used under certain assumptions in the same way as modules in other languages. For instance, several Java tools perform whole-package static analysis, although a change in any file of the package may invalidate the results of the entire analysis.

cannot be parameterized, while a class can. This prevents us from using mixin-like patterns with C++ namespaces. With class nesting and parameterized inheritance, mixin layers are a kind of module with some desirable characteristics from a software engineering standpoint.

5.3 Meta-Object Protocols

Meta-Object Protocols (e.g., [Forman et al. 1994; Kiczales et al. 1991]) are reflective facilities for modifying the behavior of an object system while the system is being used. Classical modifications include executing arbitrary code around method invocations (method *wrapping*) and changing the semantics of inheritance. Specific examples of method wrapping include function tracing, invariant checking, and object locking [Forman et al. 1994].

Meta-object protocols solve a different problem than mixin layers. Mixin layers address the issue of grouping class refinements together so they can be treated as a unit. In contrast, meta-object protocols can express modifications to fundamental operations of an object system. Meta-object protocols can be used for desirable functionality additions that are not convenient with mixin layers—e.g., the application of a single wrapper to all methods of a class at once. Of course, a meta-object protocol is a mechanism, not a design guideline. An appropriately designed meta-object protocol, allowing the encapsulation of many metaclasses in parameterized modules, could certainly be used to implement mixin layers. Unfortunately, to our knowledge, none of the standard meta-object protocols offer such encapsulation capabilities.

5.4 Aspect-Oriented Programming

Aspect-oriented programming (AOP) advocates decomposing application domains into orthogonal *aspects* [Kiczales et al. 1997]. Aspects are distinct implementation entities that encapsulate code which would otherwise be intertwined throughout an application. In this respect, aspect-oriented programming seems strikingly similar to GenVoca. Indeed, early AOP manifestos [Kiczales et al. 1997] are very similar to the work describing GenVoca generators: the software engineering arguments are identical and the implementation techniques used are very similar. Many of the AOP example applications in [Kiczales et al. 1997] are layered generators for domain-specific languages (an image processing language, a language for specifying data transfer on remote procedure calls, etc.). Domain-specific languages (or language extensions) are called *aspect languages* in AOP terminology and generators are called *aspect weavers*.

An aspect, just like a collaboration, expresses a refinement that affects multiple classes of an application. In this sense, mixin layers can be regarded as an aspect-oriented implementation technique. Nevertheless, it is perhaps hard to find cross-cutting software implementation techniques that would *not* qualify as "aspect-oriented". The term has nowadays acquired broad meaning and encompasses many different techniques. We view using "aspect-oriented" terminology as purely a matter of taste. Certainly, the cross-cutting software development ideas pre-date the introduction of "aspect-orientation".

5.5 Adaptive OO Components

Another approach to modular OO software development is Lieberherr's *Demeter* method and adaptive components [Lieberherr 1996; Lieberherr and Patt-Shamir 1997; Mezini and Lieberherr 1998]. Adaptive components specify functionality additions based on an abstract pattern of participating classes. The pattern can later be applied to actual classes of an application to extend their capabilities. This technique is analogous to identifying collaborations in an object-oriented design, only now collaborations are implementation-level entities. Note that mixin layers offer the same flexibility through the concept of adaptor layers discussed in Section 3.2.1. An important difference is that adaptor layers are themselves mixin layers. That is, with mixin layers, both the representation of a collaboration and the representation of a collaboration application are the same (namely, mixin layers).

Nevertheless, the work on adaptive components reveals an interesting direction of research, with no counterpart in our work. Adaptive components can be declared by a *strategy.* That is, a strategy is a way to declaratively specify a path through the *class graph* (the graph induced on classes by inheritance and containment relationships among them). Along each node in the strategy, extra methods can be added. In this way, strategies are compact ways of expressing functionality additions to many classes. For example, one can easily specify new methods to be added to a class *and all its superclasses.* Similarly, assume that class `A` has a member variable that can hold an instance of class `B`, which, in turn, may hold an instance of class `C`. Using strategies, a programmer can describe the path from `A` to `C` in the class graph. (Class `B` does not need to be specified explicitly.) An adaptive component employing this strategy can then define a new method to be added to all three classes. Thus, strategies are a higher-level way of specifying collaborations (refinements); mixin layers could be used to implement strategies.

5.6 Design Patterns for Modularization

The *visitor* design pattern [Gamma et al. 1995] serves similar modularization purposes to mixin layers. Visitor is a pattern allowing a *functional* style of programming in object-oriented languages: multiple definitions of the same operation (applicable to objects of several different classes) can be grouped together in a visitor class, instead of these methods being distributed over individual classes. Visitor is a fundamental modularization mechanism and has been used to implement more sophisticated techniques (e.g., [Mezini and Lieberherr 1998]).

Visitors are different from mixin layers in two ways. First, visitors are dynamic in nature, whereas mixin layers are static. This means that mixin layers can be used to add state (i.e., member variables) to the classes they refine. (For instance, imagine a class describing a graph node. If one wants to maintain the information "`is_marked`" for all nodes, this is easier to do with mixin layers: an `is_marked` field can be added in a mixin and carried in every single refined node object. With a visitor-based approach, this information must be maintained in a table on the side.) Additionally, visitors impose a run-time overhead, unlike mixin layers. Second, visitors are not allowed to access the internals of the classes they extend. In contrast, mixin layers define subclasses of the refined classes. Hence, mixin layers are often able to access more implementation details than visitors. For instance, a C++

class may export a fairly extensive interface to its subclasses (using the `protected` keyword), without making the same interface public so that visitors can use it. This issue commonly arises when other design patterns (e.g., *singleton*) are used in conjunction with the visitor pattern.

Visitors, like many other design patterns, express refinements of objects or classes. Although not a design pattern, a mixin layer can be viewed as an elegant way of expressing a collaboration pattern among classes so that it is clear at the language level. Mixin layers can be expressed with the aid of a type system, rather than bypassing it, so that more compile-time checking and optimization is possible.

5.7 Subjectivity

Objects written for one application may not be reusable in another because their interfaces are different, even though both applications may deal with what is fundamentally the same object. The principle of *subjectivity* asserts that no single interface can adequately describe any object; objects are described by a family of related interfaces [Harrison and Ossher 1993; Ossher and Harrison 1992; Ossher et al. 1995]. The appropriate interface for an object is application-dependent (or *subjective*).

Subjectivity arose from the need for simplifying programming abstractions—e.g., defining views that emphasize relevant aspects of objects and that hide irrelevant details. Ossher and Harrison took an important step further by recognizing that application-specific views of inheritance hierarchies can be produced automatically by composing different "subjects" [Harrison and Ossher 1993]. Subjects encapsulate a primitive aspect or "view" of a hierarchy, whose implementation requires a set of additions (e.g., new data and method members) to one or more classes of the hierarchy.

Collaboration-based designs and mixin layers are analogous to subjectivity and subjects. Nevertheless, even though the goals are common, different parts of the problem are emphasized in the two approaches. The biggest difference between subject-oriented programming and our approach is that a subject-oriented approach aspires to combine programs that are developed completely independently. Mixin layers focus on a different problem: the consistent refinement of groups of classes, in order to raise the level of programming from single-class to multiple-class components. Mixin layers need to be developed with interoperability in mind. This makes mixin layers a more general technique, but with a lower degree of automation and little applicability to pre-written software—manual adaptation is required.

6. CONCLUSIONS

Improved modularizations are the key to improved component-based software development. We and others have observed that traditional notions of modularization—method, class, package—are inadequate for this purpose. Many different results in modularization point to large-scale refinements—the ability to encapsulate and modularize fragments of classes and methods—as the basis for next-generation modularizations. The core idea centers on the idea of refinement as the centerpiece for component-based software development. Our refinements are large-scale: a single refinement can update multiple classes of an application, and a composition of a few refinements specifies a complete implementation of an application.

The fragments of classes and methods that need to be encapsulated are *not arbitrary.* Rather, fragments are encapsulated together when they all define how a particular service or feature, which can be shared by many applications of a domain, is implemented. That is, these fragments must have meaningful expressions in software designs. We have shown that the object-oriented concept of collaboration based designs captures this idea. A collaboration is an abstract design that specifies roles for different classes of objects, and defines protocols by which objects of these classes interact to realize a particular service or feature. Collaborations are the way large-scale (i.e., multi-class) refinements are expressed in object-oriented models. Applications are typically defined by compositions of a small number of reusable collaborations.

We have shown how collaborations can be defined and composed statically using existing programming language constructs, and how they can be supported by new language constructs. We presented a particular way of expressing large-scale refinements as *mixin layers*, a name chosen to emphasize its connection to the common *mixin* concept in object-oriented languages. We showed how mixin layers overcame the scalability difficulties that plagued prior work. They rely on a novel combination of parameterized inheritance and class nesting, in effect generalizing the concept of a package (set of classes) so that parameterized packages could participate in inheritance lattices. As an example, we showed how mixin layers were used as the primary implementation technique for building an extensible compiler for the Java language.

REFERENCES

AGESEN, O., FREUND, S. N., AND MITCHELL, J. C. 1997. Adding type parameterization to the java language. In *Conference Proceedings of OOPSLA '97, Atlanta.* ACM SIGPLAN Notices, vol. 32(10). ACM, 49–65.

BATORY, D., CARDONE, R., AND SMARAGDAKIS, Y. 2000. Object-oriented frameworks and product lines. In *Proceedings of the First Software Product Line Conference*, P. Donohoe, Ed. 227–247.

BATORY, D. AND GERACI, B. 1997. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering 23,* 2 (Feb.), 67–82.

BATORY, D., JOHNSON, C., MACDONALD, B., AND VON HEEDER, D. 2000. Achieving extensibility through product-lines and domain-specific languages: A case study. In *Proceedings of the Sixth International Conference on Software Reuse*, W. B. Frakes, Ed. 117–136.

BATORY, D., LOFASO, B., AND SMARAGDAKIS, Y. 1998. JTS: Tools for implementing domain-specific languages. In *Proceedings: Fifth International Conference on Software Reuse*, P. Devanbu and J. Poulin, Eds. IEEE Computer Society Press, 143–153.

BATORY, D. AND O'MALLEY, S. 1992. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology 1,* 4 (Oct.), 355–398.

BATORY, D., SINGHAL, V., SIRKIN, M., AND THOMAS, J. 1993. Scalable Software Libraries. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering.* 191–199.

BATORY, D. S. 1987. Concepts for a database system synthesizer. In *Symposium on Principles of Database Systems (PODS '88).* ACM Press, New York, 184–192.

BATORY, D. S., BARNETT, J. R., GARZA, J. F., SMITH, K. P., TSUKUDA, K., TWICHELL, B. C., AND WISE, T. E. 1988. GENESIS: An extensible database management system. *Software Engineering 14,* 11, 1711–1730.

BIGGERSTAFF, T. J. 1994. The library scaling problem and the limits of concrete component reuse. In *Proceedings: 3rd International Conference on Software Reuse*, W. Frakes, Ed. IEEE Computer Society Press, 102–109.

Bracha, G. and Cook, W. 1990. Mixin-based inheritance. In *OOPSLA/ECOOP '90 Proceedings*, N. Meyrowitz, Ed. ACM SIGPLAN, 303–311.

Bracha, G. and Griswold, D. 1996. Extending smalltalk with mixins. Workshop on Extending Smalltalk at OOPSLA 1996. See `http://java.sun.com/people/gbracha/mwp.html`.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, C. Chambers, Ed. ACM SIGPLAN Notices volume 33 number 10. Vancouver, BC, 183–200.

Cardelli, L. and Wegner, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys 17,* 4 (Dec.), 471–522.

Chiba, S. 1996. Open C++ programmer's guide for version 2. Tech. Rep. SPL-96-024, Xerox PARC.

Coglianese, L. and Szymanski, R. 1993. DSSA-ADAGE: An environment for architecture-based avionics development. Proceedings of the NATO AGARD Conference.

Cunningham, W. and Beck, K. 1989. Constructing Abstractions for Object-Oriented Applications. Journal of Object-Oriented Programming, July 1989.

Findler, R. B. and Flatt, M. 1998. Modular object-oriented programming with units and mixins. In *International Conference on Functional Programming (ICFP '98)*. 94–104.

Flatt, M., Krishnamurthi, S., and Felleisen, M. 1998. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*. ACM, New York, NY, 171–183.

Forman, I. R., Danforth, S., and Madduri, H. 1994. Composition of before/after metaclasses in SOM. In *Proceedings of OOPSLA'94*. ACM Sigplan Notices, vol. 29. Portland, 427–439.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns*. Addison Wesley, Reading, MA.

Habermann, A. N., Flon, L., and Cooprider, L. W. 1976. Modularization and hierarchy in a family of operating systems. *Communications of the ACM 19,* 5 (May), 266–272.

Harrison, W. and Ossher, H. 1993. Subject-oriented programming (A critique of pure objects). In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*. ACM Press, Los Alamitos, CA, USA, 411–28.

Heidemann, J. S. and Popek, G. J. 1994. File-system development with stackable layers. *ACM Transactions on Computer Systems 12,* 1 (Feb.), 58–89.

Helm, R., Holland, I. M., and Gangopadhyay, D. 1990. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*. 169–180. Published as ACM SIGPLAN Notices, volume 25, number 10.

Holland, I. M. 1992. Specifying Reusable Components Using Contracts. In *Proceedings of the ECOOP '92 European Conference on Object-oriented Programming*, O. L. Madsen, Ed. LNCS 615. Springer-Verlag, Utrecht, The Netherlands, 287–308.

International Organization for Standardization 1995. *Ada 95 Reference Manual. The Language. The Standard Libraries.* International Organization for Standardization. ANSI/ISO/IEC-8652:1995.

Johnson, R. E. and Foote, B. 1988. Designing reusable classes. *Journal of Object-Oriented Programming 1,* 2, 22–35.

Keller, R. and Hölzle, U. 1998. Binary component adaptation. In *ECOOP '98—Object-Oriented Programming*, E. Jul, Ed. Lecture Notes in Computer Science, vol. 1445. Springer, 307–329.

Kiczales, G., des Rivieres, J., and Bobrow, D. G. 1991. *The Art of the Meta-Object Protocol.* MIT Press, Cambridge (MA), USA.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming,*

M. Akşit and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 1241. Springer, 220–242.

Lieberherr, K. and Patt-Shamir, B. 1997. Traversals of object structures: Specification and efficient implementation. Tech. Rep. NU-CCS-97-15, College of Computer Science, Northeastern University.

Lieberherr, K. J. 1996. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company.

Madsen, O. L. and Møller-Pedersen, B. 1989. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA'89 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, N. Meyrowitz, Ed. ACM Press, 397–406.

Mezini, M. 1997. Dynamic object evolution without name collisions. In *ECOOP'97—Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 1241. Springer, 190–219.

Mezini, M. and Lieberherr, K. 1998. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*. ACM SIGPLAN Notices, vol. 33, 10. ACM Press, New York, 97–116.

Milner, R., Tofte, M., and Harper, R. W. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts.

Montlick, T. 1996. Implementing mixins in smalltalk. The Smalltalk Report, July 1996.

Moon, D. A. 1986. Object-oriented programming with *flavors*. In *OOPSLA'86 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, N. Meyrowitz, Ed. ACM SIGPLAN, ACM Press, 1–8.

Myers, A. C., Bank, J. A., and Liskov, B. 1997. Parameterized types for Java. In *Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, ACM Press, 132–145.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Paris, France, 146–159.

O'Malley, S. W. and Peterson, L. L. 1992. A dynamic network architecture. *ACM Transactions on Computer Systems 10,* 2 (May), 110–143.

Ossher, H. and Harrison, W. 1992. Combination of Inheritance Hierarchies. In *Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications*. 25–40. Published as ACM SIGPLAN Notices, volume 27, number 10.

Ossher, H., Kaplan, M., Harrison, W., Katz, A., and Kruskal, V. 1995. Subject-oriented composition rules. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 235–250.

Parnas, D. L. 1979. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering SE-5,* 2 (Mar.), 128–38.

Reasoning Systems 1990. *Dialect User's Guide*. Reasoning Systems.

Reenskaug, T., Anderson, E., Berre, A., Hurlen, A., Landmark, A., Lehne, O., Nordhagen, E., Ness-Ulseth, E., Oftedal, G., Skaar, A., and Stenslet, P. 1992. OORASS: Seamless support for the creation and maintenance of object-oriented systems. *Journal of Object-Oriented Programming 5,* 6 (Oct.), 27–41.

Rumbaugh, J. 1994. Getting started: Using use cases to capture requirements. *Journal of Object-Oriented Programming 7,* 5 (Sept.), 8–23.

Smaragdakis, Y. 1999. Implementing large-scale object-oriented components. Ph.D. thesis, University of Texas at Austin.

Smaragdakis, Y. and Batory, D. 1998. Implementing layered designs with mixin layers. In *Proceedings ECOOP'98*, E. Jul, Ed. LNCS 1445. Brussels, Belgium, 550–570.

Smaragdakis, Y. and Batory, D. 2000. Mixin-based programming in C++. In *GCSE'00—Generative and Component-Based Software Engineering Symposium*. Lecture Notes in Computer Science, vol. 2177. Springer, 163–177.

Steyaert, P., Codenie, W., D'Hondt, T., Hondt, K. D., Lucas, C., and Limberghen, M. V. 1993. Nested Mixin-Methods in Agora. In *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, O. Nierstrasz, Ed. LNCS 707. Springer-Verlag, Kaiserslautern, Germany, 197–219.

Stroustrup, B. 1997. *The C++ Programming Language*, 3 ed. Addison-Wesley, Reading, Mass.

Sun Microsystems 1997. *Java Inner Classes Specification.* Sun Microsystems. In `http://java.sun.com/products/jdk/1.1/docs/`.

Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr, S. M. 1999. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE'99.* Los Angeles CA, USA, 107–119.

Thorup, K. K. 1997. Genericity in Java with virtual types. In *ECOOP'97—Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 1241. Springer, 444–471.

VanHilst, M. 1997. Role-oriented programming for software evolution. Ph.D. thesis, University of Washington, Seattle, Washington.

VanHilst, M. and Notkin, D. 1996a. Decoupling change from design. In *SIGSOFT'96: Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, D. Garlan, Ed. ACM Press, 58–69.

VanHilst, M. and Notkin, D. 1996b. Using C++ Templates to Implement Role-Based Designs. In *JSSST International Symposium on Object Technologies for Advanced Software.* Springer Verlag, 22–37.

VanHilst, M. and Notkin, D. 1996c. Using role components to implement collaboration-based designs. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.* ACM Press, 359–369.

Villarreal, E. 1994. Automated compiler generation for extensible data languages. Ph.D. thesis, University of Texas at Austin.

Weihe, K. 1997. A software engineering perspective on algorithmics. In `http://www.informatik.uni-konstanz.de/Preprints/`.

Weise, D. and Crew, R. 1993. Programmable syntax macros. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation.* 156–165.

Weiss, D. M. 1990. Synthesis operational scenarios. Tech. Rep. 90038-N, Version 1.00.01, Software Productivity Consortium, Herndon, Virginia.

Wile, D. 1993. Popart: Producer of parsers and related tools. Tech. rep., USC/Information Sciences Institute.

Wirth, N. 1977. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM 20,* 11 (Nov.), 822–823.

...

Design Wizards and Visual Programming Environments for GenVoca Generators

Don Batory, *Member*, *IEEE Computer Society*, Gang Chen, Eric Robertson, and Tao Wang

Abstract—Domain-specific generators will increasingly rely on graphical languages for declarative specifications of target applications. Such languages will provide front-ends to generators and related tools to produce customized code on demand. Critical to the success of this approach will be domain-specific design wizards, tools that guide users in their selection of components for constructing particular applications. In this paper, we present the P3 ContainerStore graphical language, its generator, and design wizard.

Index Terms—Self-adaptive software, architectural optimizations, generators, components, refinements, applications product-lines.

1 Introduction

Domain-specific languages (DSLs) will become progressively more important as a medium for specifying customized applications [5], [22], [15], [34]. Generators are tools—compilers, really—that convert DSL application specifications into optimized source code. Visual programming languages, such as Visual Basic and VisualAge, will simplify the use of DSLs and promote their promulgation. But, more importantly, visual programming (or, more accurately, visual specification) languages will offer a convenient way to integrate a suite of analysis tools that will substantially enhance the capabilities and effectiveness of generators.

We are exploring the use of a visual specification and analysis environment for a Java-based generator called P3. P3 is a GenVoca (i.e., component-based) generator for container data structures that is a successor to P2 [5], [6]. P3 is a modular extension of the Java language that allows container data structures to be specified declaratively. That is, P3 adds data-structure-specific statements to Java so that users can compactly specify the implementation of a target data structure as a composition of reusable P3 components. The P3 generator, which is actually a Java preprocessor, translates P3 programs directly into pure Java programs. Among the features that make P3 attractive is that it is equivalent to a gargantuan library of container data structures whose efficiency is comparable to (or better than) hand-coded libraries that are now available.

To promote and simplify the use of P3, we have developed the ContainerStore applet as a visual programming language for writing P3 programs. Clients fill in forms and edit diagrams from which the ContainerStore can infer P3 data structure specifications. While the applet itself is not a major innovation, it is interesting because it integrates a suite of tools and services that makes P3 programming more effective—*tools and services that P3 alone cannot provide*. The ContainerStore offers facilities for *explaining* compositions of components so that clients can verify that the data structure that they have defined is the one that they want. If there are errors in a component composition (or in any other phase of specification), they are caught immediately and explanations of how to repair the errors are provided. By far, the most innovative aspect of the ContainerStore tool suite is a prototype technology for automatically critiquing and optimizing container implementations (i.e., P3 component compositions) for a particular workload. Given a set of components and rules that express knowledge of what combinations of components are best suited for solving particular problems, a tool called a *design wizard* applies these rules automatically to critique and optimize a P3 specification. If the design wizard discovers a composition of components that is likely to perform better than that specified by a user, this composition is reported and reasons are given to explain why the alternative composition is an improvement. In this way, design wizards offer expert guidance so that design blunders can be avoided.

In this paper, we present the P3 ContainerStore applet, its generator, and its design wizard.

2 The P3 ContainerStore Applet

The *ContainerStore* is a visual domain-specific language for specifying container data structures. Specifications are disributed across five tabs (i.e., presentation windows/panels) and are completed in sequence (see Fig. 1). The first tab allows a user to specify the class of elements that are to be stored. In particular, the name of the element class, and the name of each attribute, its type, and cardinality are entered.[1] As a running example, suppose elements of class **emp** are to be stored, where **emp** objects have **name** and **age** attributes.

• *The authors are with the Department of Computer Science, University of Texas at Austin, Austin, TX 78712. E-mail: batory@cs.utexas.edu.*

Manuscript received 26 Oct. 1998; revised 29 Apr. 1999; accepted 13 May 1999.
Recommended for acceptance by D.E. Perry.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 110649.

[1] The *cardinality* of an attribute is the expected number of distinct values that the attribute will be assigned.

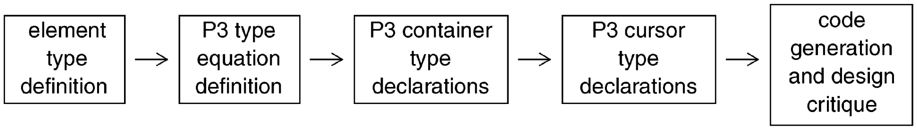

element type definition
P3 type equation definition
P3 container type declarations
P3 cursor type declarations
code generation and design critique

Fig. 1. Sequence of panel specifications in ContainerStore.

The second tab—called the *Type Equation Tab* (Fig. 2)—presents a visual interface for defining customized container implementations as a linear composition of P3 components (also known as a *P3 type equation* or *P3 component stack*). Fig. 2 shows two stacks: the "**Equation**" stack has components **rbtree** (red-black tree) on top of **hash** (hash table) on top of **malloc** (transient heap); the "**Equation2**" stack has **dlist** (doubly linked list) on top of **hashcmp** (hash comparison) and **malloc**. Stacks can be edited (e.g., components can be replaced and deleted) and annotations can be added. An *annotation* is a configuration parameter that is specific to a component. In Fig. 2, annotations to the **hash** data structure component are specified by clicking **hash** and entering the name of the key to hash (**age**) and the number of buckets (**100**) in the *Annotations* fields.

Once a type equation (component stack) has been constructed, the *Explain/View Type Equation* button is pressed. If the equation is valid, an explanation of its meaning is shown in the *Explain Window*. In Fig. 2, the meaning of the "**Equation**" stack is:

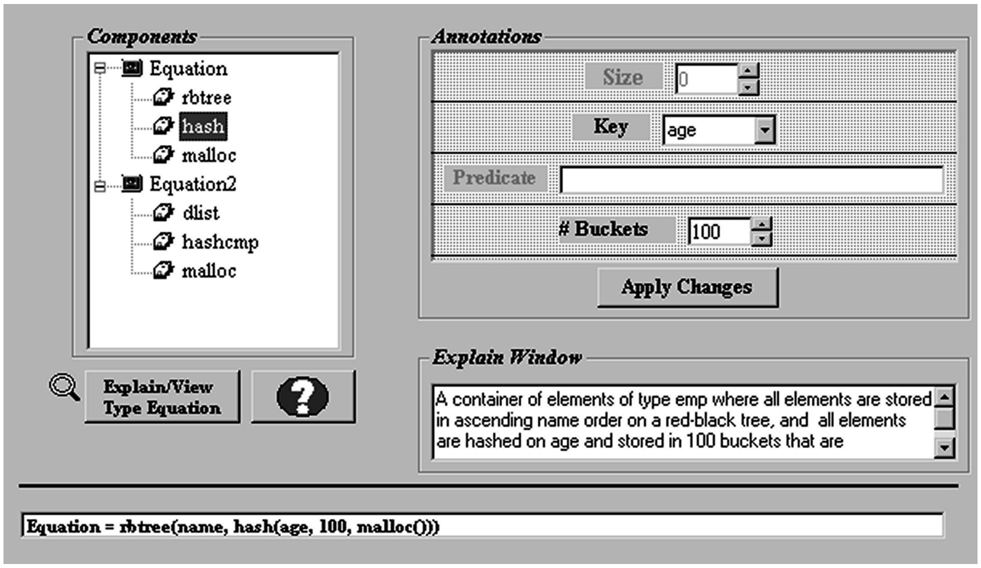
> A container of elements of type emp where all elements are stored in ascending name order on a red-black tree and all elements are hashed on age and stored in 100 buckets that are insertion-ordered doubly linked lists in transient memory.

As a general rule, people who are unfamiliar with P3 type equations are unfamiliar with their interpretation. For these users, this facility for explaining equations is invaluable. Explanations are generated using the same techniques that P3 uses to generate Java code; that is, instead of composing code fragments, the explanation is composed from English phrases. In the case that an equation is incorrect—i.e., constraints for the correct usage of a component have been violated (see [7])—the Explain Window lists the errors and suggests reparations. For example, "**Equation2**" is incorrect:

Design Rule Error: move hashcmp above dlist;
Design Rule Error: no retrieval layer beneath hashcmp;

The first error message says that ordering of the **hashcmp** and **dlist** layers is incorrect; a correct ordering would reverse their positions in the equation (the actual reasons are low-level and are not given—only that a correct composition requires **hashcmp** to be above **dlist**). Applying this modification (it turns out) satisfies the objections of the second error message, thus yielding a correct equation. In this way, the ContainerStore gives invaluable guidance to software designers: It helps them repair incorrect compositions and it helps them verify that the specified data structure is indeed the one that they want.

The third tab is where the names of the container classes and their implementing type equations are specified. Suppose container class named **ec** implemented by **Equation** is defined. The fourth tab specifies cursor classes (Fig. 3). A *cursor* is a run-time object that is used to reference, update, and delete elements in a container. In Fig. 3, the cursor class **few** is defined. Its constructor has a single parameter **x** of type **ec**—meaning that every **few** instance will be bound to an instance of container class **ec**. The selection predicate of a cursor class is specified incrementally using the *Predicate Builder*, which allows clauses of the form (attribute relation value) to be declared

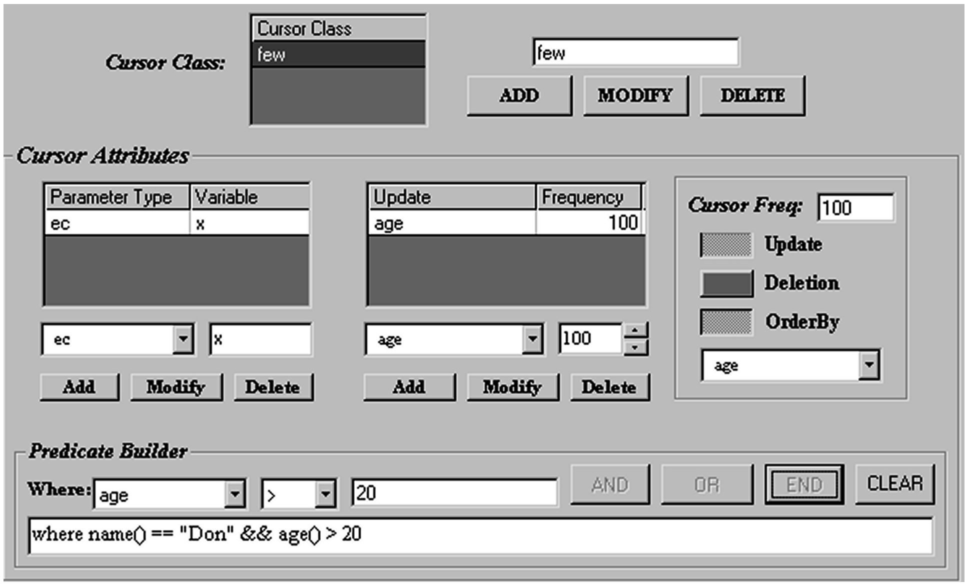


Size: 0
Key: age
Predicate:
Buckets: 100

A container of elements of type emp where all elements are stored in ascending name order on a red-black tree, and all elements are hashed on age and stored in 100 buckets that are

Equation = rbtree(name, hash(age, 100, malloc()))

Fig. 2. The Type Equation Tab.



Cursor Class: few

Parameter Type	Variable
ec	x

Update	Frequency
age	100

Cursor Freq: 100

▨ Update

■ Deletion

▨ OrderBy

age

Where: age > 20

where name() == "Don" && age() > 20

Fig. 3. The Cursor Tab.

separately. The predicate for **few** selects **ec** container elements where attribute **name** is "**Don**" and **age** is greater than **20**. In addition, elements will be retrieved in **age** order (as specified by the *OrderBy* field) and the **age** attribute of selected elements will be updated. The frequency that **few** retrievals are performed is estimated by the user to be **100** times per time period.

Finally, the fifth tab (the Generate tab) presents a scrollable window and four buttons. One button generates the P3 specification of a ContainerStore declaration; a second button invokes the P3 generator to convert this specification into Java source. The generated source is displayed in the scrollable window. (Section 3 illustrates both specifications and generated source). The third button generates a workload specification, which lists each cursor and container operation that is to be performed with its execution frequency. (This information was collected while instantiating previous tabs). The last button sends this specification to ContainerStore's design wizard to be analyzed and critiqued for its efficiency. (Section 5 elaborates this workload specification and analysis).

3 The P3 Generator

The *Jakarta Tool Suite* (JTS) is a set of domain-independent tools for building extensible domain-specific languages and GenVoca (component-based) generators [9]. JTS is written in Jak, an extensible superset of the Java language. Jak minimally extends Java with the addition of metaprogramming features (e.g., syntax tree constructors) so that Java programs can create and manipulate other Java programs. JTS is itself a GenVoca generator, where variants of Jak are assembled from components. One component in the JTS library encapsulates the P3 generator and the P3 component library.

Like its predecessor P2, P3 allows users to specify containers and cursors declaratively, just like database systems provide declarative specifications of relations and relational queries. In particular, P3 adds cursor and container data structure declarations to Java. Examples of these declarations are listed on the lefthand side of Fig. 4; the generated Java code is shown on the right. When Jak parses each of these declarations, it generates the

```
(1) container< emp > empcont;     - - - - - - - ->  interface empcont { ... }

(2) cursor< empcont > empcursor;   - - - - - - ->  interface empcursor { ... }

(3) container ec implements empcont   - - - - - ->  class ec implements empcont { ... }
       using odlist( age, malloc( ) );

(4) cursor all(ec e);          - - - - - - - - - ->  class all implements empcursor {
                                                        all(ec e) { ... }
                                                        ... }

(5) cursor few(ec e)           - - - - - - - - - ->  class few implements empcursor {
       where name() == "Don" && age() > 20              few(ec e) { ... }
       orderby age;                                     ... }
```

Fig. 4. P3 declarations and generated classes.

TABLE 1
P3 Data Structure Components

DS Component	Semantics
`malloc`	elements are stored in transient memory
`persistent`	elements are stored in persistent memory
`odlist( key x, DS y )`	elements are stored on an **x**-ordered doubly-linked list$^{\#}$
`dlist( DS y )`	elements are stored on an unordered doubly-linked list$^{\#}$
`rbtree( key x, DS y )`	elements are stored on a red-black tree with key **x**$^{\#}$
`predindex( predicate p, DS y )`	elements that satisfy predicate **p** are linked on a separate doubly-ordered linked list$^{\#}$
`hashcmp( key x, DS y )`	equality predicates on key **x** are hashed to improve performance$^{\#}$
`hash( key x, int n, DS y )`	elements are hashed on **x** and stored in a hash table with **n** buckets; each bucket implemented by a doubly-linked list$^{\#}$
`bstree( key x, DS y )`	elements are stored on a binary tree with key **x**$^{\#}$

*Note: parameter **y** defines a stack of components that lie below the given component in a P3 type equation.*

appropriate Java interface or class definition and replaces the declaration with the generated code.

Suppose instances of class **emp** are to be stored in a container. Lines (1) and (2) in Fig. 4 concisely declare Java interfaces for containers (**empcont**) and cursors (**empcursor**) that are specialized for **emp** instances. Note that the C++-like syntax for these declarations was chosen deliberately to indicate that container interfaces are parameterized by the elements (**emp**) to be stored and cursor interfaces are parameterized by the container (**empcont**) over which cursor instances will range.

Among the methods in the **empcont** interface (not shown in Fig. 4) are **emp** instance insertion and a test for container overflow. Among the methods in the **empcursor** interface are positioning a cursor on the first **emp** instance of a container, advancing to the next **emp** instance, testing for end-of-container, and get and set methods for each attribute of **emp**.

Each container class is declared separately. Statement (3) defines a container class **ec** that implements the **empcont** interface by storing **emp** instances in an age-attribute-ordered doubly linked list in transient memory. **odlist(age,malloc())** is its P3 type equation that defines both the stacking of components (i.e., **odlist** sits atop of **malloc**) and annotations (i.e., **age** is the key of the **odlist** data structure). Table 1 lists the library of components that P3 currently offers.

It is worth noting that P3 components are *not* the same as traditional parameterized components for data structures (e.g., STL [27]), but instead are "refinements" [33], [5], [6]. To see the difference, consider the interpretation of the composition of hash and bintree:

hash[keyA, 400, bintree[keyB, ...]]

This composition does *not* mean that a container of elements is hash-partitioned into **400** buckets on **keyA** and then each bucket is implemented as a binary tree ordered on **keyB**. The correct interpretation is much closer to the way database systems compose access methods: Each container element is simultaneously linked onto two distinct and independently traversable data structures—a hash structure on **keyA** and a binary tree on **keyB**. Thus, one can access every element of the container by traversing either data structure, where, obviously, the hash structure provides faster access to elements on **keyA**, while the binary tree provides faster access via **keyB**. More generally, P3 allows elements to be linked into *any* number of distinct data structures, thus enabling P3 users to define arbitrarily complicated structures that simply could not be created using traditional parameterized components. This distinction is explained in greater detail in [33], [5, [6].

Each P3 cursor class is also declared separately. Statement (4) declares a cursor class **all** of whose instances return every element of an **ec** container. The syntax of the statement defines the arguments of the constructor of the generated class (i.e., each **all** instance is bound to a particular **ec** container instance). P3 infers that **all** implements the **empcursor** interface (because **ec** containers store **emp** instances and cursors over **ec** containers implement the **empcursor** interface).

Statement (5) declares another cursor class **few** whose instances return in attribute **age** order only those elements of an **ec** container where **name == "Don"** and **age > 20**. (Again, each instance of few is bound to a single **ec** container and only returns instances of that container that satisfy the **few** predicate). Other features of P3 that are not shown in Fig. 4 include parameterized selections (i.e., **city() == x**, where **x** is specified at run-time) and declarations of cursor usage (e.g., retrieval only, element modification/deletion) for optimizing generated code.

TABLE 2
Code Size of Dictionary Benchmark Programs (in Words)

	dlist	bstree	rbtree	hash
JDK	541	N/A	N/A	506
CAL	540	561	561	562
JGL	534	N/A	N/A	540
PIZZA	565	N/A	N/A	509
P3	570	572	572	574

TABLE 3
Execution Times of Dictionary Benchmark Programs (in secs)

	dlist	bstree	rbtree	hash
JDK	82.5*	N/A	N/A	8.2
CAL	117.4	19.4	17.3	13.5**
JGL	116.9	N/A	N/A	8.1
PIZZA	99.2 ***	N/A	N/A	8.7
P3	74.9	13.8	12.8	7.9

*JDK Vector data structure is used here.
** CAL does not have explicit support for hash tables; the Set container is used instead. Internally, CAL implements Set by hash table.
*** Pizza Vector data is used here.

4 Performance of P3-Generated Code

Generators, contrary to handwritten component libraries, offer a scalable way to produce customized software [5], [11]. The declarative way in which P3 users specify container and cursor implementations through component composition leads to huge families of customized data structures. As shown in [8], significant increases in productivity and major cost reductions both in maintenance and experimentation with different container implementations can result from generators. While we have not yet used P3 in a sophisticated Java application, we have performed preliminary benchmarks on P3-generated code to assure us that P3 is on a trajectory that is comparable with its predecessors [33], [5], [6]. In this section, we review some of our preliminary results on P3's performance.

The *Container and Algorithm Library* (CAL) [39] and the *Java Generic Collection Library* (JGL) [18] are two popular and publicly available Java data structure libraries. Both are based on STL [27] and are optimized for performance. *Pizza* (a dialect of Java that supports parametric polymorphism [29]) and Sun's *Java Development Kit* (JDK) also provide simple data structures, so we also included them in our study. Presently, CAL and JGL support features (adaptors for stacks, queues, etc.) that P3 does not yet offer. (Adaptors can be encapsulated as P3 components that will be stacked on top of P3 containers to give them noncontainer interfaces; so, there is no a priori reason why such capabilities/componentry cannot eventually be added to P3).

We performed a number of experiments that benchmarked productivity and performance; the most revealing of which are presented in Tables 2 and 3. The benchmark of [5] was used to evaluate the performance of the Booch Components, libg++, and the P1 and P2 generators. We used this program for our studies. The program spell-checks a document against a dictionary of 25,000 words. The main activities are inserting randomly ordered words of the dictionary into a container, inserting words of the target document into a second container and eliminating duplicates, and printing those words of the document container that do not appear in the dictionary. The document that we used was the Declaration of Independence (~1,600 words).

We used JDK, CAL, JGL, Pizza, and P3 to implement this program using four different container implementations: doubly linked lists, binary search trees, red-black trees, and hash tables. The benchmarks were executed on a Pentium Pro 200 with 64 MB of memory, running Windows NT Workstation version 4.0. The programs were compiled and executed using JDK version 1.1.3, with the **-O** optimization option. We also recompiled the CAL beta 2 and JGL 2.0.2 libraries using JDK version 1.1.3 to ensure the validity of comparison.

Table 2 shows the program sizes for different libraries. (Sizes were obtained by removing comments and using the Unix **wc** utility to count the words). P3 programs are slightly longer than the corresponding CAL, JGL, Pizza, and JDK programs because P3 declarative specifications are more verbose than class references to Java packages. Such differences are not significant because P3 can generate vast numbers of data structures that have no counterpart in the CAL, JGL, Pizza, and JDK libraries. In such cases, these libraries would not be of much help as the target data structure would have to be written by hand. The brevity of the corresponding P3 programs and the speed at which their Java source is produced would be unchallenged. So too would the ability to alter container implementations quickly and easily (by merely redefining the P3 type equation and recompiling); significantly more work would be needed using Pizza, CAL, JGL, and JDK.

Table 3 lists the execution times for each program. In general, P3 programs outperform their hand-coded counterparts for two reasons. First, both CAL and JGL are based on STL, but, since Java does not support templates, both have to rely extensively on inheritance. This introduces additional dispatches and down-casts, which slows execution. Second, and more significant, there is inherent overhead in the JDK, CAL, Pizza, and JGL designs. These libraries are designed for *generic* applications, whereas the programs generated by P3 are produced for a *specific* task. Consider element comparisons. P3 directly inlines comparison expressions, whereas CAL and JGL programs have to use a "predicate" object that encapsulates a function to evaluate that predicate on a given element. (This is a common way to work around the lack of function pointers in Java.) Note that this function can*not* be optimized by the Java compiler because it knows nothing about query optimization. There are other inefficiencies that preclude significant optimizations that generators can provide.

The point of our experiments was to provide minimal confirmation that P3 generates code comparable to that written by hand. Clearly, many more experiments are needed. We have no illusions that this simple example is sufficient in any way; our goal at this stage of our research is to demonstrate that the performance of P3-generated code conforms to that observed in earlier generators, which it does.

5 The P3 Design Wizard

A fundamental problem in all component-based generators is: Given a workload specification and a set of components, how should one select and assemble components to define an appropriate application implementation? In the case of P3, what type equation (data structure) would efficiently process a given workload? This is a difficult problem for two reasons.

First, software designers are rarely aware of the actual workload that an application will subject a data structure. A designer will know the kinds of queries asked (e.g., since these queries will be specified as **cursor** declarations), but the actual frequency with which particular **cursor** classes are instantiated and elements are retrieved will not be known until run-time. At best, only educated guesses can be made (and, often, these estimates are determined instinctively).

Second, even if a workload is known precisely, it can be a challenging problem to determine an efficient data structure. When a workload is simple, the problem is easy. For example, if elements of a container are to be accessed only via the predicate **N == <value>**, then a hash table with elements hashed on field **N** is likely to be an optimal choice. However, if workloads become slightly more complicated, it is hard to tell what data structure would be best. For example, if there are 20,000 elements, 3,000 elements are inserted and deleted per time period, fields **S** and **N** are updated 1,000 times per period, elements are retrieved using predicate **N == <value> && A==“b”** 2,000 times per period, and all elements are retrieved in **S** order 50 times per period, what data structure would most efficiently support this workload? The answer is not obvious even to experienced programmers.

To solve the first problem, one can instrument generated code so that it collects workload statistics at run-time. So, initially, one fields an application knowing full well that its data structures are not optimal. After a period of time, enough statistics will have been collected so that a more appropriate data structure can be determined. The application **cursor** and **container** classes are then regenerated and the old classes discarded. A new cycle of collect-statistics-and-regenerate then begins. Normally, a programmer is in the loop to close the cycle (i.e., a programmer decides how long to collect statistics, how to use these statistics to deduce a better data structure, and when to initiate the class regeneration and replacement). However, this loop could be closed *without* programmer intervention. That is, the *application* determines when enough statistics have been collected, a tool called a *design wizard* finds a more efficient data structure given this workload, and if regeneration is warranted, class regeneration and replacement is performed automatically. Such software is called *self-adaptive* [23], [36], [30], [2] and may be the ultimate way to minimize software development and maintenance costs through component reuse.

The key to achieving self-adaptive software requires a solution to the second problem—deducing an efficient type equation for given a workload. This requires a kind of knowledge that is not present in GenVoca domain models (and domain models in general). Knowledge of when and how to use a component effectively to maximize performance or to meet a design objective is quite different than that of design rules (i.e., requirements that define the correct usage of a component [7]). What form this knowledge will take, what is a general model to express such knowledge, and how to optimize type equations remain open problems. Short of proposing a general-purpose theory, it is possible to develop ad hoc techniques for given domains and, in particular, for data structures. By abstracting from specific solutions in different domains (i.e., performing a “domain analysis” on these solutions), a general theory may result.

For now, however, we outline an approach that we have found effective to optimize and critique P3 type equations automatically given a workload specification. While the solution itself is domain-specific, it does constitute a valuable first step toward self-adaptive software and a general model of design wizards.

5.1 P3 Workload Specifications

Data structure optimization is a well-studied problem. Because P3 presents a relational-like interface to data structures, relational database optimization models are an obvious starting point (e.g., [26]). A *workload* on a database relation (or P3 container) is characterized by the type and cardinality of individual attributes of an element, plus the frequency with which each container or cursor operation is performed. Fig. 5 illustrates a workload specification file produced by the ContainerStore applet. (The information of Fig. 5 was provided by a ContainerStore client when he/she filled in the operation frequency slots in the ContainerStore specification tabs or it might be the result of a statistical analysis of running instance of the target application.) It states that there are 5,000 elements in a container. Each element has two fields, one is a String called **name** that has 5,000 unique values, etc. Three hundred elements are inserted per time period, all elements are retrieved in **name** order 100 times per period, and so on. The type equation (which implements the container whose workload is defined in Fig. 5) that is to be critiqued is **odlist(age,malloc())**.

5.2 Cost Model

Given a workload W and a container implementation (type equation) T, we want to estimate the cost of processing W using T. This is accomplished by synthesizing cost functions.[2] The cost function we seek, $Cost(T, W)$, is the sum of

[2] The method by which cost functions are produced is exactly the same as that used in P3 code synthesis and type equation explanations.

```
workload {
    cardinality = 5000;

    attributes {
        #id      type     cardinality
        #--------------------------
        name     String   5000;
        age      int      60;
    }

    work {
        #operation        frequency
        #--------------------------
        insertion         300;
        deletion          300;
        ret orderby name  100;
        ret where name() == "Don" &&
            age() > 20
            orderby age   100;
    }

    Equation = odlist(age, malloc());
}
```

Fig. 5. P3 workload specification.

the costs of processing each individual cursor and container operation times its execution frequency. The cost of an individual operation is the sum of the costs contributed by individual layers of T. For example, every layer performs some action when an element is inserted into a container. Thus, the cost of an element insertion is equal to the sum of the costs of insertion actions that are performed by each layer (see Fig. 6). The same holds for attribute update and element deletion. Retrieval costs are estimated a bit differently, as query optimization is involved. A retrieval predicate is processed by traversing a single data structure. The data structure that is to be traversed (i.e., the structure whose traversal algorithms are to be generated) is the one that returns the minimum cost estimate for processing that predicate. This "polling" of layers/data structures by P3 is called *query optimization*. Selected functions that define $Cost(T, W)$ are summarized in Table 4, where n denotes the number of elements in a container, b the number of hash buckets, and c is a constant. Different data structures will have different values for c for different operations, where particular values are determined by benchmarking P3 data structures on a specific platform.[3,4]

$$Cost(T, W) = I(T) \times InsFreq + D(T) \times DelFreq +$$
$$\sum_j (U(T, Field_j) \times UpdFreq_j) + \sum_j (R(T, Ret_j) \times RetFreq_j)$$
$$I(T) = \sum_{i \in T} insertionCost(layer_i)$$
$$D(T) = \sum_{i \in T} deletionCost(layer_i)$$
$$U(T, Field_j) = \sum_{i \in T} updateCost(layer_i, Field_j)$$
$$R(T, Ret_j) = Min_{i \in T}(retrieval(layer_i, Ret_j))$$

Fig. 6. P3 cost model.

$Cost(T, W)$ again is used to evaluate a particular design T for a workload W. Ideally, a design wizard must walk the space of all legal type equations and find the equation T that minimizes $Cost(T, W)$. In the next section, we explain how this space is defined and, later, how our wizard walks this space.

5.3 The Space of P3 Type Equations

P3 components are characterized by three kinds of attributes: properties, signatures, and design rules. Together they define the space of all syntactically and semantically correct P3 type equations. A *Layer Declaration File* (LDF) is a specification of this information, an example of which is shown in Fig. 7.

Properties are attributes that classify components [7]. In Fig. 7, six different properties are defined. **logical_key** is the propositional symbol for the attribute that defines "a key-ordered component," i.e., a P3 component that implements a data structure that stores elements in key order. Red-black trees and ordered doubly linked lists have this property. Similarly, **hash_key** is the propositional symbol for the attribute that defines "a hash component," i.e., a P3 component that implements a data structure that stores elements via hashing. As we will see shortly, properties are used to express both design rules and type equation rewrite rules (discussed in the next section). Consistent with the experience discussed in [7], determining these properties is a fairly straightforward task.

Signatures define the export and import interfaces of a component; these properties are used to determine if a component usage in a type equation is syntactically correct. In Fig. 7, **ds = {...}** denotes the usual GenVoca syntax for a *realm* (i.e., library) of components that implement the interface **ds** [Bat92]. Three such components are listed: **rbtree**, **delflag**, and **malloc**. The signature of the **rbtree** (red-black) tree is circled. **rbtree** has a **keyfield** parameter and **ds** parameter (which means that **rbtree** can be composed with other **ds** components). In contrast, the **malloc** component has no parameters.

Not all syntactically correct type equations are semantically correct. Domain-specific constraints called *design rules* are needed to define the legal uses of a component. The algorithms that we use for design rule checking are given in [7]. Design rules are expressed in two parts. First, properties that are asserted or negated by a component are broadcast

[3] *Equality retrieval* are predicates of the form **key == value**; *range retrieval* are predicates of the form **low-value < key < high-value**; *scan retrievals* do not qualify elements on key values.

[4] Although we list only the higher-order terms in Table 4, we included lower-order terms in our prototype. It turns out that basic problems which plague database researchers for obtaining accurate estimates of query processing costs also hinder us (see [31]). For example, it is well-known that accurate estimates of query and subquery selectivity are difficult to obtain. While we could use more advanced techniques of estimation, experience has shown that this is overkill for typical P3 applications. When designing data structures, most programmers do not sit down with a calculator to determine the most efficient data structure; rather, they apply heuristics learned in their data structure courses to design and implement their container. These heuristics are captured by these equations when n, the number of elements in a container, is "sufficiently large." When the number of elements is not known (which is generally the case), these heuristics are reasonable. See [5], [6] for examples.

TABLE 4
Selected Individual Cost Equations

Layers	insertion	deletion	update	equality retrieval	range retrieval	scan retrieval
dlist	*c*	*c*	*c*	*c*n*	*c*n*	*c*n*
rbtree	*c*log(n)*	*c*log(n)*	*key: c*log(n)* *non-key: c*	*key: c*log(n)* *non-key: c*n*	*key: c*log(n)* *non-key: c*n*	*c*n*
hash	*c*	*c*	*key: c* *non-key: c*	*key: c(n/b)* *non-key: c*n*	*c*n*	*c*n*

to all layers that lie above it and below it in a type equation. These properties are declared by the **asserted properties** and **negated properties** statements. For example, the **malloc** component broadcasts the asserted property **transmem** when it is used in a type equation. Similarly, the **rbtree** component broadcasts the asserted properties **retrieval** and **logical_key**.

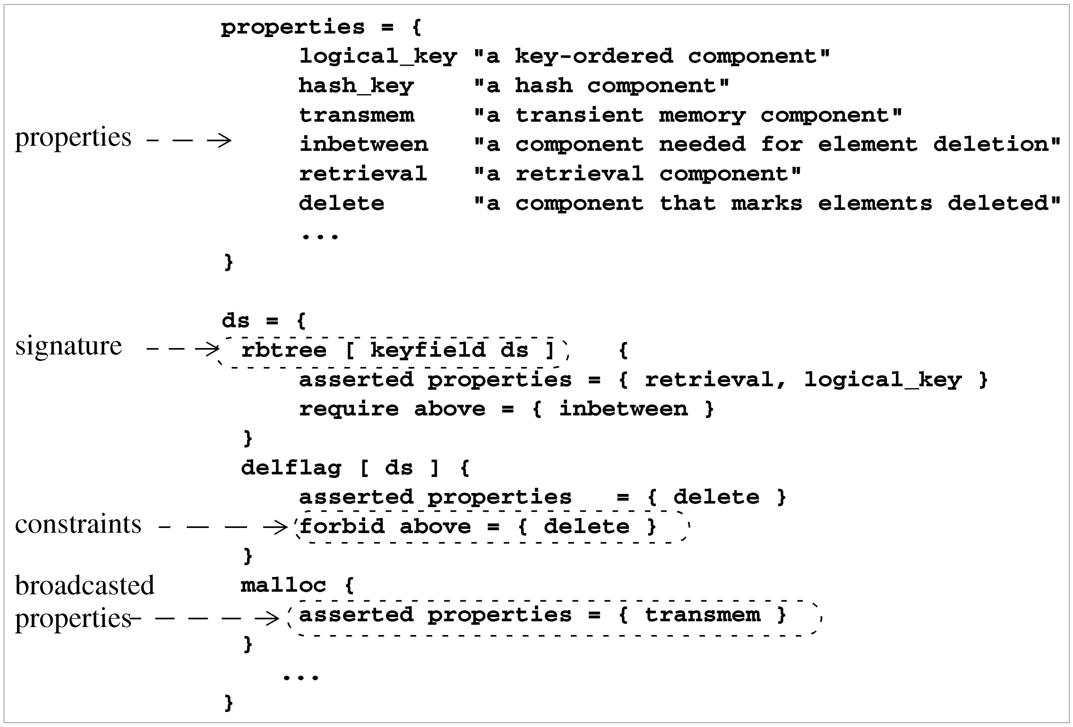
Second, preconditions for component usage are expressed as conjunctive predicates. Asserted properties are expressed with the **require** statement; negated properties with the **forbid** statement. Thus, if a component **X** has the declarations:

require above = { A, B }
forbid above = { C }

they define the predicate **A** ∧ **B** ∧ ¬**C** which must be satisfied by layers that lie *above* **X** in a type equation. By replacing "**above**" with "**below**," the predicate must be satisfied by layers that lie *below* **X** in a type equation. (Thus, different conditions can be imposed on layers above **X** and below **X** in an equation). As an example that combines both property broadcasting with preconditions, the **delflag** layer in Fig. 7 allows only one instance of itself in a type equation. That is, the first **delflag** instance will broadcast the **delete** property, while a second **delflag** will detect its presence when its precondition ¬**delete** fails.

5.4 Automatic Optimization of Equations

The space of all P3 type equations is the set of all design-rule-correct type equations that can be composed using the given components. The size of this space is enormous: If there are k components in the P3 library, the number of type equations with c components is $O(k^c)$. So, even for small k and c, an exhaustive search is infeasible. While the number of components in an equation is theoretically unbounded, we know from experience that domain experts can quickly identify an efficient equation with few (i.e., typically under 10) components.



```
properties = {
     logical_key "a key-ordered component"
     hash_key    "a hash component"
     transmem    "a transient memory component"
     inbetween   "a component needed for element deletion"
     retrieval   "a retrieval component"
     delete      "a component that marks elements deleted"
     ...
}

ds = {
  rbtree [ keyfield ds ]   {
     asserted properties = { retrieval, logical_key }
     require above = { inbetween }
  }
  delflag [ ds ] {
     asserted properties   = { delete }
     forbid above = { delete }
  }
  malloc {
     asserted properties = { transmem }
  }
    ...
}
```

Fig. 7. A P3 layer declaration file.

The number of equations that are relevant to an application is a subspace of the entire P3 space. One way in which this subspace can be generated is by applying rewrite rules that transform one equation into an equivalent equation, starting from an initial feasible solution/equation (e.g., [17]). The rewrite rules that we use are derived from an analysis of the heuristics that we have personally applied to produce efficient type equations manually. In particular, our intuitive optimization strategy has been guided by three heuristics:

- When an equation rewrite is attempted, we check that the resulting equation is *consistent* (i.e., it is syntactically correct and it satisfies the design rules);
- The cost of the rewritten equation is unchanged or lowered;
- Rewrites are considered in an order (we feel) will most likely lead to a new equation with lower cost.

Some of our rules deal with element attributes. Consider the following rewrite that is expressed in two parts:

1. If an element attribute **A** is listed as an **orderby** key in the workload specification, then try to insert a **logical_key** layer (such as a red-black tree or an ordered-list) with **A** as its key.

The idea of this rewrite is that it is cheaper to store elements in sorted order rather than sorting an unordered set of elements on demand. This rewrite may fail if there already exists a **logical_key** layer with that attribute as key. (The reason for failure is that the $Cost(T, W)$ of the rewritten equation T will be higher—the rationale is that a single data structure that maintains element order is usually cheaper than two structures maintaining the same order.) This leads to the second part of the rewrite:

2. If 1 fails, then try to replace the **logical_key** layer with **A** as its key with a more efficient **logical_key** layer.

The idea of this rewrite is that if there already exists a data structure that maintains elements in key order, there may be a more efficient data structure to accomplish the same task. This rewrite attempts to find a such a replacement.

Readers may have observed the use of Layer Declaration File properties in expressing rules. To apply the above rule, our design wizard searches its library for components that assert the **logical_key** property. These components are candidates for insertion or replacement in the above rule. Different rules qualify components on different properties. Consider a second rewrite:

- If element attribute **A** is used in an equality retrieval predicate (e.g., **name == "Don"**), then try to insert a **hash_key** component with **A** as its key; if there already exists such a layer, try to substitute it with a more efficient **hash_key** layer.[5]

These and similar rules are growth rules—i.e., they add components to type equations. There are growth rules that do no involve element attributes. There are also *shrink rules*—i.e., rules that remove components from type equations. An example is:

- Remove a component from a type equation if it increases *Cost*.

The optimization of a P3 type equation is similar to an AI planning process [16]. We discovered that optimizing P3 equations manually followed a best-first (greedy) heuristic; we automated this search to find a correct and efficient type equation with regard to the given workload, cost models, and layer declarations. Because the equations that are retained in the search have progressively lower cost, we are guaranteed to find a local minimum. The search can begin from scratch, starting from a trivial data structure—such as a doubly linked list in transient memory. However, when used with the ContainerStore applet, the search begins with the type equation that was specified in the workload.

Overall, we have about 10 different rewrite rules. The basic algorithm that we use to apply these rewrites to optimize type equations is:

```
for each element attribute A {
  apply each "attribute growth"
    rewrite for A;
}
apply each "nonattribute growth" rewrite;
apply each "shrink" rewrite;
```

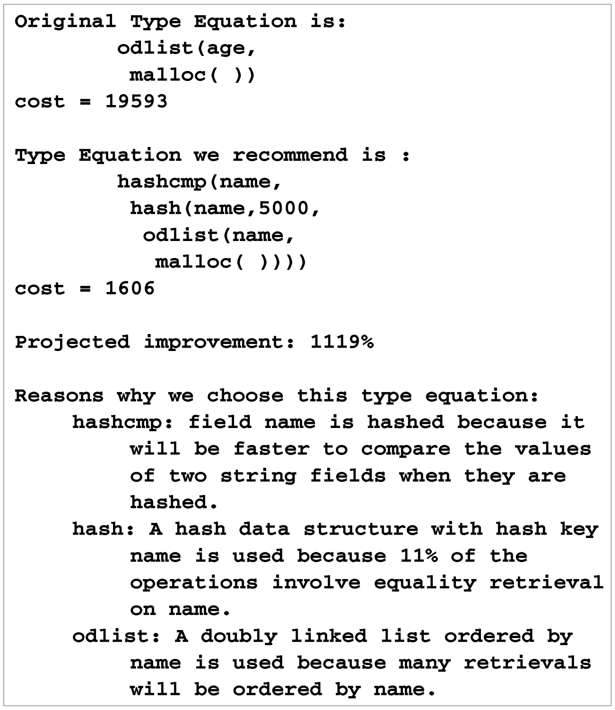
The algorithm is run to a fixpoint (i.e., the algorithm is continually invoked until no further rewriting is possible) and, thus, will identify a local minimum. If the P3 subspace defined by our rewrite rules is well-formed (i.e., has only one minimum), our algorithm is guaranteed to find it. If the subspace has several local minima, our algorithm will locate one, but not necessarily the global minimum. We are unaware of any theoretical result that would tell us whether a P3 subspace is (or is not) well-formed. Lacking such information, it is possible that a more powerful search algorithm might uncover better results. However, the results we have obtained using this algorithm have been quite good—occasionally better, but never worse than, what we would have manually selected. Moreover, we are unaware of a proposed equation that we could subsequently improve. Although much more work (e.g., using more powerful search algorithms) remains, we believe that a greedy search algorithm is a reasonable first step.[6]

5.5 Critique and Optimization

Given a workload specification, the P3 design wizard applies its rewrites to the input type equation. If there is

[5] At present, P3 has only one hash component. A component for dynamic hashing may be added later.

[6] Since the conference publication of this paper, we have a proof that the P3 search space can indeed have multiple local minima and that finding the global minimum is NP-hard. An example that illustrates the problem involves processing every query of a set of queries using some keyed data structure (e.g., binary-tree). Suppose that creating a keyed data structure on field A, then another on field B, and a third for field C will allow all queries in the set to be processed efficiently by traversing one of these structures. Call these structures the ABC indexing set. Now, suppose that we had created a keyed structure on field E and a second on field F and all queries of the set could be processed efficiently by traversing either the E structure or F structure. Call this the EF indexing set. Clearly, the ABC indexing set and the EF indexing set are local minima. If our design wizard selects the ABC as an answer, it will be unable to "backtrack" to find solution EF (or vice versa) and, hence, will not find the global minimum.



```
Original Type Equation is:
        odlist(age,
         malloc( ))
cost = 19593

Type Equation we recommend is :
        hashcmp(name,
         hash(name,5000,
          odlist(name,
           malloc( ))))
cost = 1606

Projected improvement: 1119%

Reasons why we choose this type equation:
    hashcmp: field name is hashed because it
         will be faster to compare the values
         of two string fields when they are
         hashed.
    hash: A hash data structure with hash key
         name is used because 11% of the
         operations involve equality retrieval
         on name.
    odlist: A doubly linked list ordered by
         name is used because many retrievals
         will be ordered by name.
```

Fig. 8. Critique and optimization of a Type Equation.

no substantial improvement, the wizard simply reports that no changes to the equation need to be made. A more likely response is that it will have discovered an implementation/equation that has better performance characteristics. Fig. 8 shows the output of a critique using the workload of Fig. 5.

Both the input and revised equations are presented, along with their cost (i.e., $Cost(T, W)$) estimates. An explanation is also presented which provides reasons why the generated equation is better. The reason is that the original data structure linked elements together onto an **age**-ordered list. The workload, on the other hand, demands that all elements of the container be periodically retrieved in **name** order and that individual elements (whose name is "**Don**") be retrieved frequently. The original data structure does not efficiently support this workload at all. The recommended data structure allows elements to be accessed quickly (via hashing) on **name** and that elements be stored in order on **name** (via an ordered linked list). Furthermore, to speed up the search for elements on **name**, the **hashcmp** component is used. (**hashcmp** transforms equality predicates on strings (**name == "Don"**) to include integer comparisons (**hash_of_name == hash("Don") && name == "Don"**). The idea is that integer comparisons are much faster than string comparisons). While most programmers would not think to add this enhancement (probably because it is tedious for a programmer to add by hand), it is quite simple for P3 to do it. The performance enhancements for altering the type equation are predicted by the design wizard to pay-off handsomely. (The actual percentage reported in Fig. 8 is not particularly important; rather, it gives users some idea of how much better the suggested design would be.)

There are two general contributions that design wizards make to automated software development. First, not all users of a generator will be domain experts. Even if they have familiarity with a domain, they may not know as much as an expert, or, in the case of design wizards, a host of domain experts. Design wizards will help avoid blunders and will help users find more efficient implementations for their target systems. Second, and possibly more significant, type equation synthesis is a prerequisite to adaptive software—applications that dynamically change their configuration as a function of current workload. For most domains—including data structures—manual reconfigurations are rarely done because of the costs involved. (The problem becomes even more complicated if data structures are persistent; updating data structures requires the additional cost of unloading data from old structures and reinserting it into the new structures). As a consequence, application users must suffer with degraded performance and application developers must endure the costs of program maintenance. Design wizards have the potential to change this situation dramatically.

6 Related Work

It is widely believed that *domain-specific languages* (DSLs) will significantly impact future software development. DSLs offer concise ways of expressing complex, domain-specific concepts and applications, which in turn can offer substantially reduced maintenance costs, more evolvable software, and significant increases in software productivity [5], [22], [15]. Generators are compilers for DSLs [34]. Component-based generators, such as P2 and P3, show how reusable components form the basis of a powerful technology for producing high-performance, customized applications in a DSL setting (see also [28]).

The automatic selection of data structures is an example of automatic programming [3]. SETL is a set-oriented language where implementations of sets can be specified manually or determined automatically [32]. SETL offers very few set implementations (bit vector, indexed set, and hashing) and relies on a static analysis of an SETL program using heuristics rather than using cost-based optimizations to decide which set implementation to use. AP5 relies more on user-supplied annotations for data structure selection [13].

Deductive program synthesis is another way to achieve automatic programming [3], [35], [25], [24], [19]. The idea is to define a domain theory (typically in first order logic) that expresses fundamental relationships among basic domain entities. A domain theory, together with a theorem prover and theorem-proving tactics, can find a constructive proof for a program specification and extract from this proof computational methods from which a program can be synthesized. Finding a proof may be fully automatic, but frequently requires guidance from users to help navigate through the space of possible proofs. Our design wizard is very different. First, finding a "proof" (a P3 type equation) for a workload specification is trivial—simply implement every container as a doubly-linked list. All container and cursor operations will be processed, but not efficiently. The challenge is finding a P3 type equation that efficiently processes that workload. Second, work on program synthesis has largely focused on generating *algorithms* (e.g., algorithms for solving PDEs [19], algorithms for scheduling

[35], algorithms for computing solar incidence angles [24], etc.); subroutines are the components from which generated algorithms are built. The inferences needed for algorithm synthesis tend to be quite sophisticated (thus requiring theorem provers) because there are very complex relationships among domain entities. In contrast, GenVoca is different both in component scale and in the simplicity of the relationships among domain entities. GenVoca components are *subsystems*—suites of interrelated OO classes. (A P3 component, for example, encapsulates three classes: a cursor class, a container class, and an element class.) As noted in [7], scaling the size of components *and* designing components to be plug-compatible has a nonobvious effect: The relationships that exist among components tend to be very simple, and elementary inferencing (i.e., no theorem provers) is adequate.

Our concept of design wizards resonates with recently proposed notions of *open implementations* (OI) [20] and *Aspect Oriented Programming (AOP)* [21]. Aspects in AOP are very similar to components in GenVoca; they encapsulate changes to be made to multiple classes when an aspect (or feature) is added to an application. Aspect weavers are functions which take a program as input and produce another (more detailed, extended, or refined) program as output. P3 components have an almost identical description. The primary difference is that AOP starts with existing application source, whereas GenVoca decomposes applications into primitive layers and reexpresses them as a compositions of these layers. P3 relies on general results from GenVoca that address the issues of optimizing across multiple layers/aspects and the order in which components/aspects can be legally composed. To our knowledge, there are no corresponding general results for AOP.

The idea of OI is that, when interfaces largely hide implementation details, it should be possible for clients to annotate abstract declarations with profiling (or other implementation-specific) information so that a compiler or server can automatically select the most appropriate implementation that is available. The OI guidelines address design issues, but implementation details are not discussed. When the space of implementations is restricted to a small handful of choices, the solution is straightforward [3], [32], [13]. However, our experience with P3 shows that declarative specifications can map to vast numbers of implementations. While design issues are indeed important, additional difficult problems remain:

1. creating a model that defines the space of possible implementations,
2. using the model to produce efficient implementations,
3. the ability to rank individual implementations in this space, and
4. efficiently walking the space.

GenVoca and design wizard provide a systematic way to address all of these concerns.

The techniques we used for optimizing type equations are very similar to those of rule-based query optimization [14], [37]. A query is represented by an expression where terms correspond to relational operators (e.g., join, sort, select). Query optimization progressively rewrites a query expression according to a set of rules, where the goal is to find the expression with the lowest cost. Since we model data structures as expressions and our design wizard progressively rewrites expressions until no further rewriting produces a more efficient expression, the problems seem identical. However, there are differences. First, constraints among relational operators can be expressed simply by algebraic rewrite rules. In contrast, we do not yet have an algebraic representation for our rules. (In fact, the implementation of our "rules" is pure Java code). Moreover, the correct usage of layers requires design rule checking, which we also have been unable to express as algebraic rewrites. Second, query optimization deals with a rather small set of operators (e.g., join, sort, select), whereas type equation optimization potentially may deal with a much larger set of operators (i.e., tens or hundreds of layers). For these reasons, type equation optimization may be more difficult than query optimization.

7 Conclusions

P3 is a GenVoca generator for container data structures. Although its basic technology was developed earlier [5], [6], P3's novelty is that it has been implemented as a modular extension to the Java language that introduces data-structure-specific statements. These statements enable P3 users to compactly and declaratively specify a family of data structures whose size dwarfs that of hand-coded Java libraries (e.g., CAL, JGL, JDK, Pizza). Besides offering broader coverage, P3 is additionally attractive because it generates efficient code. The basic reason for its efficiency—beyond the fact that the generation techniques are powerful—is that P3 produces data structures for a specific application (where all kinds of optimizations can be performed), whereas conventional libraries only offer generic data structures (where these optimizations have not been applied).

The P3 generator, however, is not sufficient for a practical software development environment. In this paper, we presented the ContainerStore applet, a visual domain-specific programming language that integrates an important suite of tools and services that P3 alone does not provide.[7] The particular services that we discussed are: English-generated explanations of P3 component compositions (which are important as P3 novices will not be familiar with component semantics), automatic validation of compositions with messages suggesting how to repair errors (if errors are detected), automatic generation of P3 code (so that users can study correct P3 specifications), automatic translation of P3 specifications into Java code (i.e., the P3 generator is called), and the automatic critique and optimization of a user-defined P3 component composition given a workload specification (i.e., the P3 design wizard is called).

Among all these services, our design wizard is the most novel. Although its optimization strategies and component rewrite rules are indeed specific to the domain of container data structures, we believe it is the first example of a much more general technology for automatic component selection and composition. The idea of optimizing component compositions by applying domain-specific rewrite rules is

[7] The capabilities described in this paper were demonstrated at the DARPA EDCS Workshop in Seattle, July 1997.

certainly not limited to container data structures. The reason is that GenVoca provides a general way in which to create vast "product-lines" from components; applications of GenVoca product-lines are expressed as type equations and the improvement of a particular equation/design is always through component replacement, insertion, and removal (i.e., equation rewrite rules).

Our initial success with the P3 design wizard is encouraging. However, it is essential that design wizards for other domains be created. We believe that analyzing design wizards for different domains may lead to a general model for expressing type equation rewrite rules. Such a model may offer a general purpose technology for achieving adaptive software—i.e., software that automatically reconfigures itself upon noticing a change in its usage/workload. Adaptive software may be the ultimate way to minimize software development and maintenance costs through component reuse.

Acknowledgments

We thank the anonymous referees who made valuable suggestions that improved the paper. We also thank Rich Cardone and Yannis Smaragdakis for their comments on earlier drafts of this paper, and Lane Warshaw for discussions that revealed the connection between index selection and multiple local minima in a P3 search space. Our Web site is http://www.cs.utexas.edu/users/schwartz/. This work was supported in part by Microsoft, Schlumberger, the University of Texas Applied Research Labs, and the US Department of Defense Advanced Research Projects Agency in cooperation with the US Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

References

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] R. Allen, R. Douence, and D. Garlan, "Specifying and Analyzing Dynamic Software Architectures," *Proc. Conf. Fundamental Approaches to Software Eng.,* Mar. 1998.

[3] R. Balzer, "A Fifteen-Year Perspective on Automatic Programming," *IEEE Trans. Software Eng.,* vol. 11, Nov. 1985.

[4] D. Batory, "On the Complexity of the Index Selection Problem," unpublished manuscript, 1978.

[5] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries," *Proc. ACM SIGSOFT,* 1993.

[6] D. Batory, J. Thomas, and M. Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler," *Proc. ACM SIGSOFT,* 1994.

[7] D. Batory and B.J. Geraci, "Validating Component Compositions and Subjectivity in GenVoca Generators," *IEEE Trans. Software Eng.,* vol. 23, no. 2, pp. 67-82, Feb. 1997.

[8] D. Batory, "Intelligent Components and Software Generators," *Proc. Software Quality Inst. Symp. Software Reliability,* Apr. 1997.

[9] D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: A Tool Suite for Building GenVoca Generators," *Proc. Fifth Int'l Conf. Software Reuse,* pp. 143-155, June 1998.

[10] I. Baxter, "Design Maintenance Systems," *Comm. ACM,* pp. 73-89, Apr. 1992.

[11] T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse" *Proc. Int'l Conf. Software Reuse,* pp. 102-110, Nov. 1994.

[12] S.V. Browne and J.W. Moore, "Reuse Library Interoperability and the World Wide Web," *Proc. Int'l Conf. Software Eng.,* pp. 684-691, May 1997.

[13] D. Cohen and N. Campbell, "Automating Relational Operations on Data Structures," *IEEE Software,* May 1993.

[14] D. Das and D. Batory, "Prairie: A Rule Specification Framework for Query Optimizers," *Proc. Int'l Conf. Data Eng.,* pp. 201-210, Mar. 1995.

[15] A. Van Duersen and P. Klint, "Little Languages: Little Maintenance?" *Proc. First ACM SIGPLAN Workshop Domain-Specific Languages,* 1997.

[16] C.M. Eastman, "Automated Space Planning," *Artificial Intelligence,* vol. 4, pp. 41-64, 1973.

[17] G. Graefe and D. DeWitt, "The Exodus Optimizer Generator," *Proc. ACM SIGMOD,* 1987.

[18] P. Jenkins, D. Whitmore, G. Glass, and M. Klobe, "JGL: The Generic Collection Library for Java," ObjectSpace Inc., URL: http://www.objectspace.com/jgl/, 1997.

[19] E. Kant et al., "Synthesis of Mathematical Modeling Software," *IEEE Software,* pp. 30-41, May 1993.

[20] G. Kiczales, J. Lampling, C.V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy, "Open Implementation Design Guidelines," *Proc. Int'l Conf. Software Eng.,* 1997.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming '97* pp. 220-242, 1997.

[22] R. Kieburtz et al., "A Software Engineering Experiment in Software Component Generation," *Proc. Int'l Conf. Software Eng.,* 1996.

[23] R. Laddaga, *Self-Adaptive Software Workshop,* Kestrel Inst., July 1997.

[24] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood, "AMPHION: Automatic Programming for Scientific Subroutine Libraries," *Proc. Int'l Symp. Methodologies for Intelligent Systems,* pp. 326-335, Oct. 1994.

[25] Z. Manna and R. Waldinger, "Fundamentals of Deductive Program Synthesis," *IEEE Trans. Software Eng.,* vol. 18, no. 8, pp. 674-704, Aug. 1992.

[26] M.F. Mitoma and K.B. Irani, "Automatic Database Schema Design and Optimization," *Proc. 1975 Very Large Databases Conf.,* pp. 286-321, 1975.

[27] D.R. Musser, A. Saini, and A. Stepanov, *STL Tutorial & Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.

[28] G. Novak, "Software Reuse by Specialization of Generic Procedures through Views," *IEEE Trans. Software Eng.,* vol. 23, no. 7, pp. 401-417, July 1997.

[29] M. Odersky and P. Wadler, "Pizza into Java: Translating Theory into Practice," *Proc. ACM Principles of Programming Languages,* 1997.

[30] P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-Based Runtime Software Evolution," *Proc. Int'l Conf. Software Eng.,* 1998.

[31] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *Proc. ACM SIGMOD 1996,* pp. 294-305, 1996.

[32] E. Schonberg, J.T. Schwartz, and M. Sharir, "An Automatic Technique for Selection of Data Representations in SETL Programs," *ACM Trans. Prog. Languages and Systems,* pp. 126-143, Apr. 1981.

[33] M. Sirkin, D. Batory, and V. Singhal, "Software Components in a Data Structure Precompiler," *Proc. 15th Int'l Conf. Software Eng.,* pp. 437-446, May 1993.

[34] Y. Smaragdakis and D. Batory, "DiSTiL: A Transformation Library for Data Structures," *Proc. USENIX Conf. Domain-Specific Languages,* 1997.

[35] D.R. Smith, "KIDS: A Semiautomatic Program Development System," *IEEE Trans. Software Eng.,* vol. 16, no. 9, pp. 1,024-1,043, Sept. 1990.

[36] J. Sztipanovits, G. Karsai, and T. Bapty, "Self-Adaptive Software for Signal Processing," *Comm. ACM,* pp. 67-73, May 1998.

[37] L. Warshaw, D. Miranker, and T. Wang, "A General Purpose Rule Language as the Basis of a Query Optimizer," UTCS TR97-19, Univ. of Texas at Austin, July 1997.

[38] J.S. Poulin and K.J. Werkman, "Melding Structured Abstracts and the World Wide Web for Retrieval of Reusable Components," *Proc. Symp. Software Reusability,* pp. 160-168, 1995.

[39] X3M Solutions, "CAL: The Container and Algorithm Library for the Java Platform,"URL: http:// www.x3m.com/products/cal/, 1997.

∗ The authors' photographs and biographies are not available at this time.

Third International Conference on Generative and □
Component-Based Software Engineering (GCSE 2001), □
September 9-13, 2001 Messe Erfurt, Erfurt, Germany.

A Standard Problem for Evaluating Product-Line Methodologies

Roberto E. Lopez-Herrejon and Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712
{rlopez,batory}@cs.utexas.edu

Abstract. We propose a standard problem to evaluate product-line methodologies. It relies on common knowledge from Computer Science, so that domain-knowledge can be easily acquired, and it is complex enough to expose the fundamental concepts of product-line methodologies. As a reference point, we present a solution to this problem using the GenVoca design methodology. We explain a series of modeling, implementation, and benchmarking issues that we encountered, so that others can understand and compare our solution with theirs.

1 Introduction

A *product-line* is a family of related software applications. A *product-line architecture* is a design for a product-line that identifies the underlying building blocks or *components* of family members, and enables the synthesis of any particular member by composing these components. Different family members (product-line applications) are represented by different combination of components. The motivation for product-line architectures is one of economics and practicality: it is too expensive to build all possible family members; it is much cheaper to build components and to assemble desired family members from them.

Many methodologies have been invented to create product-line architectures (e.g., [2, 3, 7, 9, 11, 12, 13, 14, 17, 20]). Unfortunately, the state-of-the-art is immature. We are unaware of any attempts to evaluate different methodologies on a common set of problems. If this were done, we would understand better the strengths and weaknesses of different methodologies. We would know when to use a particular methodology, and when not to. Further, we would know if different methodologies relied on the same concepts. For example, different OO design approaches rely on a common conceptual foundation of classes, interfaces, and state machines, but offer different ways of producing a design expressed in terms of these concepts. For product-line methodologies, we generally do not know even this. Different methodologies have rather different meanings for the terms “architecture”, “component”, and “composition” so that it is not at all obvious what, if anything, is in common. It is not evident that the same concepts are shared among product-line methodologies, let alone knowing what these concepts are. From a practical standpoint, the choice of which methodology to use in a situation is dictated by convenience (at best) or by random selection (at worst) rather than by scientific fact. This is unacceptable.

For this area to mature, it is essential that we compare and evaluate proposed methodologies. The scientific principles that underlie this area must be identified and the contributions and novelties of different methodologies be exposed in a way that all can appreciate and recognize. The immaturity of this area is not unique and has occurred in other areas of Computer Science. In such cases, a standard problem has been proposed and different authors have applied their methodologies to solve it (e.g., [1]). Doing so exposes important details that would otherwise be overlooked or misunderstood. Such studies allow researchers to more accurately assess the strengths, benefits, commonalities, and variabilities of different methodologies. We believe this approach would be beneficial for product-lines.

In this paper, we propose a standard problem for evaluating product-line methodologies. We believe a standard problem should have the following characteristics:

- It draws on common knowledge from Computer Science, so that the often difficult requirement of becoming a domain expert or acquiring domain-expertise is minimized.
- It is not a trivial design problem; it is complex enough to expose the key concepts of product-lines and their implementation.

These characteristics should enable others to see the similarities and differences among approaches both at a superficial level and more importantly, at a deeper conceptual level.

To carry this idea forward, we present as reference point a solution to this problem using the GenVoca design methodology. We outline a set of design, implementation, and benchmarking issues that we had to resolve before we settled on our final design. Doing so exposed a variety of concerns and insights that we believe others would benefit hearing. Our designs, code, and benchmarks are available at a web site (`http://www.cs.utexas.edu/users/dsb/GPL.html`) for others to access.

2 A Standard Problem: The Graph Product Line

The *Graph Product-Line (GPL)* is a family of classical graph applications that was inspired by early work on software extensibility [16, 19]. GPL is typical of product-lines in that applications are distinguished by the set of features that they implement, where no two applications have the same set of features.[1] Further, applications are modeled as sentences of a grammar. Figure 1a[2] shows this grammar, where tokens are names of features. Figure 1b shows a GUI that implements this grammar and allows GPL products to be specified declaratively as a series of radio-button and check-box selections.

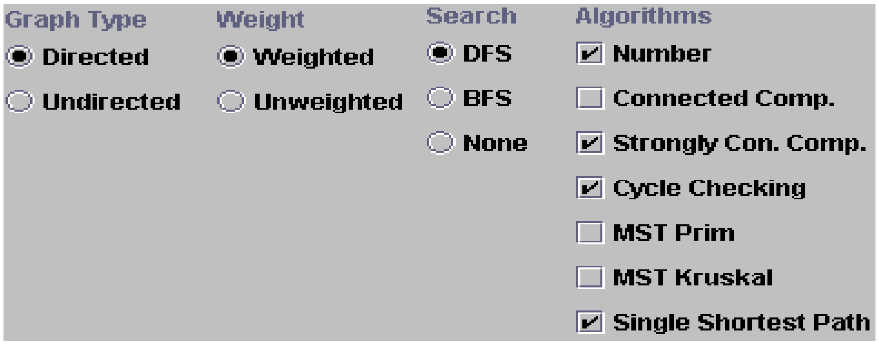
[1] A *feature* is a functionality or implementation characteristic that is important to clients [15].

[2] For simplicity, the grammar does not preclude the repetition of algorithms, whereas the GUI does.

(a)

```
GPL := Gtp Wgt Src Alg+;

Gtp := Directed | Undirected;

Wgt := Weighted | Unweighted;

Src := DFS | BFS | None;

Alg := Number | Connected | StronglyConnected
     | Cycle | MST Prim | MST Kruskal | Shortest;
```

(b)



Graph Type
● Directed
○ Undirected

Weight
● Weighted
○ Unweighted

Search
● DFS
○ BFS
○ None

Algorithms
☑ Number
□ Connected Comp.
☑ Strongly Con. Comp.
☑ Cycle Checking
□ MST Prim
□ MST Kruskal
☑ Single Shortest Path

Figure 1. GPL Grammar and Specification GUI

The semantics of GPL features, and the domain itself, are uncomplicated. A graph is either `Directed` or `Undirected`. Edges can be `Weighted` with non-negative numbers or `Unweighted`. Every graph application requires at most one search algorithm: breadth-first search (`BFS`) or depth-first search (`DFS`); and one or more of the following algorithms [10]:

- **Vertex Numbering (`Number`):** Assigns a unique number to each vertex as a result of a graph traversal.
- **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 an equivalence class, there is a path from x to y.
- **Strongly Connected Components (`StronglyConnected`):** Computes the *strongly connected components* of a directed graph, which are equivalence classes under the reachable-from relation. A vertex y is reachable form 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 directed graphs must have at least 2 edges, while in undirected graphs it must have at least 3 edges.

- **Minimum Spanning Tree (`MST Prim`, `MST Kruskal`):** 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. We include both algorithms because they present distinct and interesting performance and design issues.
- **Single-Source Shortest Path (`Shortest`):** Computes the shortest path from a source vertex to all other vertices.

A fundamental characteristic of product-lines is that not all features are compatible. That is, the selection of one feature may disable (or enable) the selection of others. GPL is no exception. The set of constraints that govern the GPL features are summarized in Table 1.

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

Table 1. Feature Constraints

A GPL application implements a valid combination of features. As examples, one GPL application implements vertex numbering and connected components using depth-first search on an undirected graph. Another implements minimum spanning trees on weighted, undirected graphs. Thus, from a client's viewpoint, to specify a particular graph application with the desired set of features is straightforward. And so too is the implementation of the GUI (Figure 1b) and constraints of Table 1.

We chose Java as our implementation language. Besides its simplicity over C++ and availability of GUI libraries, we made use of Java containers, iterators, and sort methods, to avoid reimplementing these low-level routines by hand. We recommend others to follow our lead to make comparisons easier.

3 GenVoca

GenVoca is a model of product-lines that is based on step-wise extension [3-6][3]. Among its key ideas is programs are values. Consider the following constants that represent programs with individual features:

```
f          // program that implements feature f
g          // program that implements feature g
```

An *extension* is a function that takes a program as input and produces an extended (or feature-augmented) program as output:

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

It follows that a multi-featured application is an *equation*, and that different equations define a set of related applications, i.e., a *product-line*, such as:

```
a1 = i(f)       // application a1 has features i and f
a2 = j(g)       // application a2 has features j and g
a3 = i(j(f))    // application a3 has features i, j, and f
```

Thus one can determine features of an application by inspecting its equation.

3.1 GPL

A GenVoca model of GPL is the set of constants and functions defined in Table 2. There are three extensions that are not visible to the GUI: `Transpose`, `Benchmark`, and `Prog`. `Transpose` performs graph transposition and is used (only) by the `StronglyConnected` algorithm. It made sense to separate the `StronglyConnected` algorithm from `Transpose`, as they dealt with separate concerns. (This means that an implementation constraint in using the `StronglyConnected` extension is that the `Transpose` extension must also be included, and vice versa). `Benchmark` contains functions to read a graph from a file and elementary timing functions for profiling. `Prog` creates the objects required to represent a graph, and calls the algorithms of the family member on this graph.

Extensions can not be composed in arbitrary orders. The legal compositions of extensions in Table 2 are defined by simple constraints called *design rules* [3] whose details we omit from this paper, but do include with our source code. Our GUI specification tool translates a sentence in the grammar of Figure 1 (in addition to checking for illegal combinations of features) into an equation. Because features are in 1-to-1 corre-

[3] A *refinement* adds implementation detail, but does not add methods to a class or change the semantics of existing methods. In contrast, *extensions* not only add implementation detail but also can add methods to a class and change the semantics of existing methods. Inheritance is a common way to extend classes statically in OO programming languages.

spondence with extensions, this translation is straightforward. For example, a GPL application app that implements vertex numbering and connected components using depth-first search on an undirected graph is the equation:

```
app = Prog( Benchmark( Number( Connected( DFS( Undirected )))))
```

Directed	directed graph	Cycle(x)	cycle checking
Undirected	undirected graph	MSTPrim(x)	MST Prim algorithm
Weighted(x)	weighted graph	MSTKruskal(x)	MST Kruskal algorithm
DFS(x)	depth-first search	Shortest(x)	single source shortest path
BFS(x)	breadth-first search	Transpose(x)	graph transposition
Number(x)	vertex numbering	Benchmark(x)	benchmark program
Connected(x)	connected components	Prog(x)	main program
StronglyConnected(x)	strongly connected components		

Table 2. A GenVoca Model of GPL

3.2 Mixin-Layers

There are many ways in which to implement extensions. We use mixin-layers [18]. To illustrate, recall the Directed program implements a directed graph. This program is defined by multiple classes, say Graph, Vertex, and Edge. (The exact set of classes is an interesting design problem which we discuss in Section 4). A mixin-layer that represents the Directed program is the class Directed with inner classes Graph, Vertex, and Edge:

```
class Directed {
   class Graph  {...}
   class Vertex {...}
   class Edge   {...}
}
```

An extension is implemented as a mixin, i.e., a class whose superclass is specified by a parameter. The depth-first search extension is implemented as a mixin DFS that encapsulates extensions (mixins) of the Graph and Vertex classes. That is, DFS grafts new methods and variables onto the Graph and Vertex classes to implement depth first search algorithms:

```
class DFS<x> extends x {
   class Graph  extends x.Graph  {...}
   class Vertex extends x.Vertex {...}
}
```

The above describes the general way in which GenVoca-GPL model constants and functions are implemented. When we write the composition `A = DFS(Directed)` in our model, we translate this to the equivalent template expression:

```
class A extends DFS<Directed>;
```

In general, there is a simple mapping of model equations to template/mixin expressions. Of course, Java does not support mixins or mixin-layers, but extended Java languages do. We used the *Jakarta Tool Suite (JTS)* to implement mixin-layers [4].

4 Graph Implementation

Designing programs that implement graph algorithms is an interesting problem. Every implementation will define a representation of graphs, vertices, edges, and *adjacency* — i.e., what vertices are adjacent (via an edge) to a given vertex. Further, there must be some way to represent annotations of edges (e.g., weights, names). We did not arrive at our final design immediately; we went through a series of designs that incrementally improved the clarity of our code, which we document in the following sections. In the process, we learned a simple rule to follow in order to simplify extension-based designs.

4.1 Adjacency Lists Representation (G)

The first representation we tried was based on a “legacy” C++ design [18, 5] that had been written years earlier and that implemented few of the extensions listed in Table 2. It consisted of 2 classes:

- `Graph`: consists of a list of `Vertex` objects.
- `Vertex`: contains a list of its adjacent `Vertex` objects.

That is, edges were implicit: their existence could be inferred from an adjacency list. Figure 2 illustrates this representation for a weighted graph. The advantage of this representation was its simplicity. It worked reasonably well for most extensions that we had to implement. However, it failed on edge annotations (e.g., weights). Because edges were implicitly encoded in the design, we had to maintain a weights list that was “parallel” to the adjacency list. While this did indeed work, our layered designs were obviously not clean or elegant — e.g., for operations like graph transposition which needed to read edge weights, and Kruskal’s algorithm which needed to manipulate edges directly. Because of these reasons, this lead us to our second design.

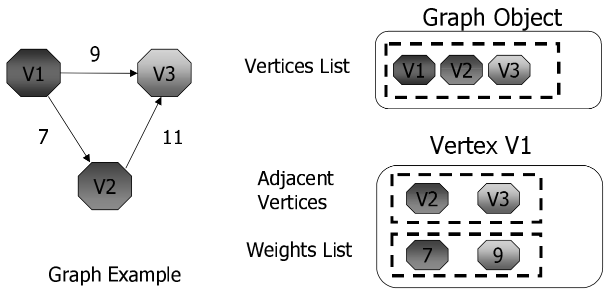


Figure 2. Adjacency Lists Representation Example

4.2 Neighbor List Representation (GN)

The second representation consisted of three classes:

- `Graph`: contains a list of `Vertex` objects.
- `Vertex`: contains a list of `Neighbor` objects.
- `Neighbor`: contains a reference to a `Vertex` object, the other end of an edge.

Edge annotations were encoded as a extensions — i.e., extra fields — of the `Neighbor` class. Figure 3 illustrates this representation. By pushing the neighbor `Vertex` object and edge annotations into a `Neighbor` object, we reduced the number of list accesses required to obtain these annotations. While this did lead to a simplification of the coding of some mixin-layers, it did not simplify the complexity of the Kruskal algorithm. Since this mixin-layer was unnecessarily difficult to write (and read!), we knew there was still something wrong. This lead to our final design.

4.3 Edge-Neighbor Representation (GEN)

Textbook descriptions of algorithms are almost always simple. The reason is that certain implementation details have been abstracted away — but this is, in fact, the strength of layers and extensions. We wanted to demonstrate that we could (almost literally) copy algorithms directly out of text books into mixin-layer code. The benefits of doing so are (a) faster and more reliable implementations and (b) easier transference of proofs of algorithm correctness into proofs of program correctness. We realized that the only way this was possible was to recognize that there are a standard set of “conceptual” objects that are referenced by all graph algorithms: Graphs, Vertices, Edges, and Neighbors (i.e., adjacencies). Algorithms in graph textbooks define the fundamental extensions of graphs, and these extensions modify Graph objects, Vertex objects, Edge objects, and Neighbor objects. Thus, the simplest way to express such extensions is to reify all of these “conceptual” objects as physical objects and give them their own distinct classes.

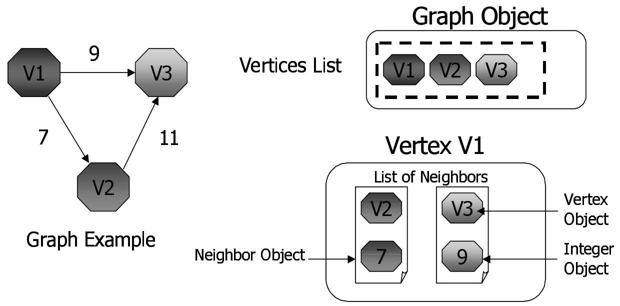


Figure 3. Neighbor Lists Representation Example

The problems of our previous designs surfaced because we tried to make “short-cuts” to avoid the explicit representation of certain conceptual objects (e.g., Edge, Neighbor). Our justification for doing so was because we felt the resulting programs would be more efficient. That is, we were performing “optimizations” in our earlier designs that folded multiple conceptual objects into a single physical object. In fact, such *premature optimizations* caused us nothing but headaches as we tried to augment our design to handle new extensions and to produce easy to read and maintain code. (We think that this may be a common mistake in most software designs, not just ours). So our “final” design made explicit all classes of objects that could be explicitly extended by graph algorithms. Namely, we had four classes:

- `Graph`: contains a list of `Vertex` objects, and a list of `Edge` objects.
- `Vertex`: contains a list of `Neighbor` objects.
- `Neighbor`: contains a reference to a neighbor `Vertex` object (the vertex in the other end of the edge), and a reference to the corresponding `Edge` object.
- `Edge`: extends the `Neighbor` class and contains the start `Vertex` of an `Edge`.

Edge annotations are now expressed as extensions of `Edge` class, and were expressed by the addition of extra fields in the `Edge` class. This representation is illustrated in Figure 4.

Equating conceptual objects with physical objects may simplify source code, but the question remains: were our original designs more efficient? Is “premature design optimization” essential for performance? These questions are addressed next.

5 Profiling Results

We performed a series of benchmarks to quantify the trade-offs between our three designs. Several implementations of the designs were tried, using different containers, and different strategies to access and copy the edge annotations. This section shows the results for our most fine-tuned implementations. The benchmarks were run on a Windows 2000 platform using a 700Mhz processor with 196MB RAM.

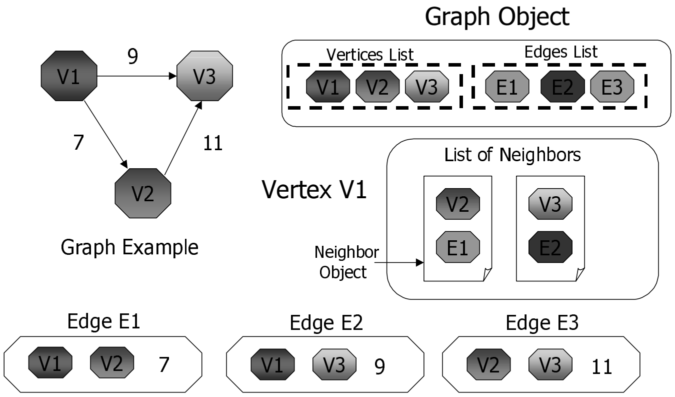
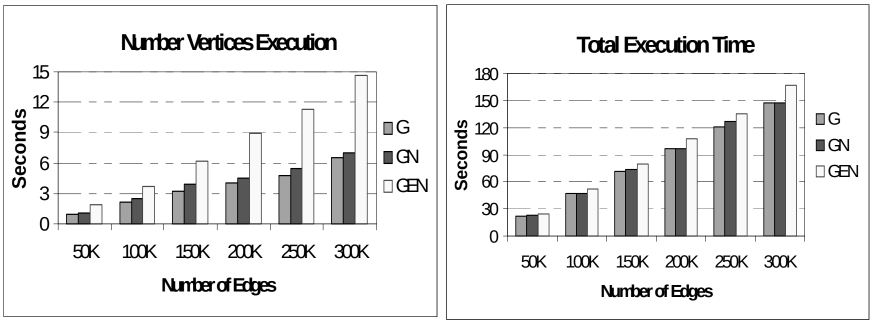


Figure 4. Edge and Neighbor List Representation Example

The first program used the vertex number algorithm on undirected graphs using depth first search. This program measured the performance of graph creation and traversal. A randomly generated graph with 1000 vertices was used as test case. Figure 5 shows the benchmark results.

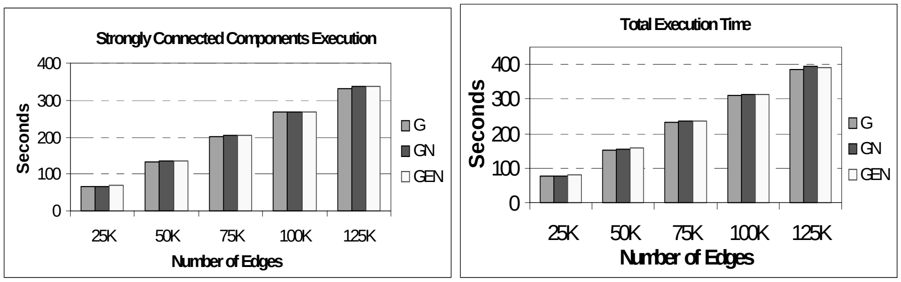
Figure 5a indicates that design **G** (our first) performs better than the other two; 6%-22% better that **GN** (our second), and 75%-120% better than **GEN** (our third). This is not surprising: **GN** and **GEN** have object creation overhead that is absent in **G** — `Neighbor` objects are created in **GN**, and `Neighbor` and `Edge` objects are created in **GEN**. While this is an obvious difference, the overall speed of the benchmark was dictated by the time reading the graph from disk. Figure 5b shows this total execution time, where the difference between the **G** application and the **GN** application is about 5% and **G** with **GEN** is about 9%.



(a) (b)

Figure 5. Simple graph traversal comparison

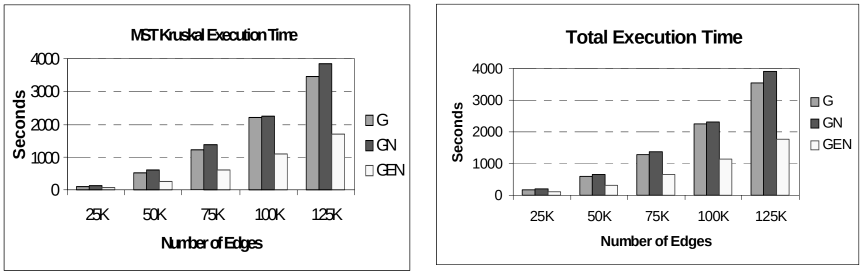
The second program benchmarked the impact of copying a graph with edge annotations. `StronglyConnected` utilizes such an operation, transpose, that creates a new copy of a graph but with the direction of the edges reversed. A randomly generated graph with 500 vertices was used as test case. In general, there was no significant difference (see Figure 6a). The **G** design performed 2% better than **GN** and 6% better than **GEN**. Although cost of graph creation is different among designs (as indicated by Figure 5a), the differences are swamped by the large computation times of the `StronglyConnected` algorithm. In particular, only 15% of the total execution time in Figure 6b was spent in reading the graph in from disk.



(a) (b)

Figure 6. Strongly Connected Components

The third program benchmarked the impact of algorithms that use edges explicitly, like Kruskal's algorithm. A randomly generated graph with 500 vertices was used as a test case. As expected, the **GEN** representation outperformed the other two simply because it does not have to compute and create the edges from the adjacency or neighbor lists. It performed between 43% and 98% faster than representation **G**, and between 59% and 120% faster than representation **GN** (see Figure 7a). The difference between **G** and **GN** is due to the fact that in the latter, to get the weights for each edge, an extra access to the weights lists is required; and that the creation of the output graph is more expensive because it has to create `Neighbor` objects as well. Of the total execution time (Figure 7b), approximately 60% was spent reading a graph of 25K edges from disk, and less than 5% when the graph had 125K edges.



(a) (b)

Figure 7. MST Kruskal

Overall, we found that the performance of algorithms that did not use weighted edges (e.g., numbering, cycle-checking, connected components, strongly-connected components) had slightly better performance with the **G** design. For those algorithms that used weighted edges (e.g., MST Prim, MST Kruskal, shortest path), the **GEN** design was better. Because an application is specified by the same equation for all three models, we could exploit our performance observations in a "smarter" generator that would decide which design/implementation would be best for a particular family member — i.e., one equation might be realized by a **G** design, another by a **GEN** design (see [6]).

Focussing exclusively on performance may be appropriate for most applications. But a more balanced viewpoint needs to consider program complexity (which indirectly measures the ease of maintenance, understandability, and extensibility). The main issue for us was the impact that the representation of edges had on program complexity. By in large, all layers had visually simple representations. But the Kruskal layer seemed more complicated than it needed to be. The reason was that in both the **G** and **GN** designs, the Kruskal layer had an explicit `Edge` class that was private to that layer, and used by no other layer[4]. (The Kruskal algorithm demanded the existence of explicit edge objects). The fact that all layers might benefit from making `Edge` explicit drove us to the **GEN** design, which we considered visually and conceptually more elegant than our earlier designs. As it turns out, our instincts on "visual simplicity" were not altogether accurate. To see why, we use two metrics for program complexity: *lines of code* (**LOC**) and *number of symbols* (**NSYMB**).[5] Table 3 shows these statistics for the Kruskal layer. Making edges explicit did indeed simplify this layer's encoding. However, other parts of our design grew a bit larger (mostly because we had to make the `Neighbor` and `Edge` classes and their extensions explicit). Table 4 shows these same statistics, across all layers, for all three designs. Overall, the statistical complexity of all three designs was virtually identical. So the drive for "visual simplicity" among layers in the end did improve our designs, but surprisingly did not impact their size statistics.

There is a benefit to the **GEN** design that is not indicated by the above tables. If we chose to enlarge the **G** and **GN** product-line with more algorithms that directly manipulate edges, then it is likely a local copy of the `Edge` class would be introduced into these layers. And doing so would result in replicated code, possibly leading to problems with program maintenance. By making the `Edge` class global to all extensions as in the **GEN** design, we would expect little or no code replication — precisely what we want in a product-line design.

Finally, we wanted to see if explicit layering (which mixin-layers produce) affects the overall performance. We created equations for each design that contained the most layers (10), and manually-inlined the resulting chain of mixin-layers into an unlayered package called *Flat*. There are two equations that have 10 layers, namely:

[4] The local version of Edge in the Kruskal layer is indicated in Table 4 as 7 lines of 52 tokens.

[5] We used other metrics [8], but found they provided no further insights.

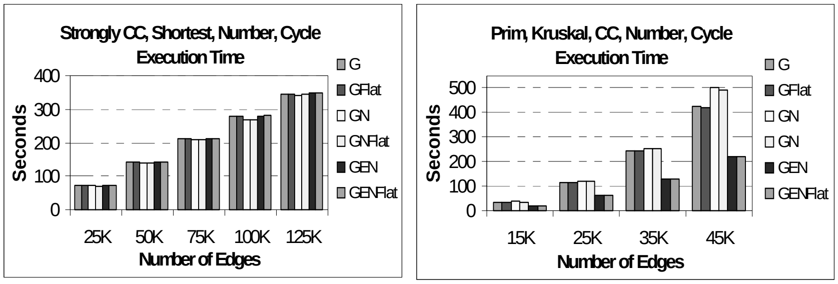
		LOC			**NSYMB**	
	G	**GN**	**GEN**	**G**	**GN**	**GEN**
Kruskal	**87**	**90**	**69**	**927**	**928**	**695**

Table 3. Kruskal Algorithm Statistics

Class		**LOC**			**NSYMB**	
Name	**G**	**GN**	**GEN**	**G**	**GN**	**GEN**
Graph	372	387	380	3554	3600	3492
Vertex	209	202	198	1832	1758	1631
Neighbor	0	30	16	0	229	49
Edge	7	7	26	52	52	304
Total	**588**	**626**	**620**	**5438**	**5639**	**5476**

Table 4. Lines of Code (LOC) and Number of Symbols (NSYMB)

- *Directed, Weighted, DFS, StronglyConnected, Number, Transpose, Shortest, Cycle, Benchmark, Prog:* in this case the difference between the layered version and the flattened one oscillates between 0% and 2% in **G**, -1% and 1% for **GN**, and -1% and 1% for **GEN**. A randomly generated graph with 500 vertices was used as test case. These results are shown in Figure 8a.
- *Undirected, Weighted, DFS, Connected, Number, Cycle, MST-Kruskal, MST-Prim, Benchmark, Prog:* for this application the difference between the layered version and the flattened one varies between 0% and 3% in **G**, 0% and 5% in **GN**, and between -1% and 1% in **GEN**. A randomly generated graph with 300 was used as test case. The results are shown in Figure 8b.



(a) (b)

Figure 8. Effect of Class Layering

The small difference between the layered version and its corresponding flattened one is due to the fact that few methods override their parent method. When overriding does occur, it involves fewer than 3 layers. Again, this result is specific to GPL and may not hold for other domains.

6 Conclusions

GPL is a simple and illustrative problem for product-line designs. Different applications of the GPL product-line are defined by unique sets of features, and not all combinations of features are permitted. The state of the art in product-lines is immature, and the need to understand the commonalities and differences among product-line design methodologies is important. We want to know how methodologies differ, what are their relative strengths and weaknesses, and most importantly what are the scientific principles that underlie these models. We do not know answers to these questions. But it is our belief that by proposing and then solving a standard set of problems, the answers to these questions will, in time, be revealed.

We believe GPL is a good candidate for a standard problem. It has the advantages of *simplicity* — it is an exercise in design and implementation that can be discussed in a relatively compact paper — and *understandability* — domain expertise is easily acquired because it is a fundamental topic in Computer Science. Further, it provides an interesting set of challenges that should clearly expose the key concepts of product-line methodologies.

In this paper, we presented a product-line model and implementation of GPL using the GenVoca methodology and the Jakarta Tool Suite (JTS). We showed how GenVoca layers correspond to features, and how compositions of features are expressed by equations implemented as inheritance lattices. We presented a sequence of designs that progressively simplified layer implementations. We benchmarked these implementations to understand performance trade-offs. As expected, different designs do have different execution efficiencies, but it is clear that a “smart” generator (which had all three designs available) could decide which representation would be best for a particular application. As an additional result, we showed that there is a very small impact of class layering in overall application performance.

We hope that others apply their methodology to GPL and publish their designs and findings. We believe that our work would benefit by a close inspection of others, and the same would hold for other methodologies as well. Our code can be downloaded from `http://www.cs.utexas.edu/users/dsb/GPL.html.`

Acknowledgements. We would like to thank Vijaya Ramachandran for her valuable help with the subtle details of the theory of graph algorithms. We also thank Jay Misra for clarifying the distinction between refinements and extensions.

7 References

[1] J-R Abrial, E. Boerger, and H. Langmaack, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, Lecture Notes in Computer Science, Vol. 1165, Springer-Verlag, 1996.

[2] P. America, et. al. “CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering”, *Software Product Lines: Experience and Research Directions*, Kluwer Academic Publishers, 2000.

[3] D. Batory and B. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, February 1997.

[4] D.Batory, B.Lofaso, and Y.Smaragdakis. “JTS: Tools for implementing Domain-Specific Languages”, *Int. Conf. on Software Reuse*, Victoria, Canada, June 1998.

[5] D. Batory, R. Cardone, and Y.Smaragdakis. “Object-Oriented Frameworks and Product Lines”, *1st Software Product-Line Conference*, Denver, Colorado, August 2000.

[6] D. Batory, G. Chen, E. Robertson, and T. Wang, “Design Wizards and Visual Programming Environments for GenVoca Generators”, *IEEE Transactions on Software Engineering*, May 2000, 441-452.

[7] J. Bosch, “Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study”, *Software Architecture*, Kluwer Academic Publishers, 1999.

[8] S.R. Chidamber and C.F. Kemerer, “Towards a Metrics Suite for Object Oriented Design”, *OOPSLA 1991*.

[9] S. Cohen and L. Northrop, “Object-Oriented Technology and Domain Analysis”, *Int. Conf. on Software Reuse*, Victoria, Canada, June 1998.

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

[11] K. Czarnecki and U.W. Eisenecker, “Components and Generative Programming”, *SIGSOFT 1999*, LNCS 1687, Springer-Verlag, 1999.

[12] K. Czarnecki and U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[13] J-M. DeBaud and K. Schmid, “A Systematic Approach to Derive the Scope of Software Product Lines”, *Int. Conference on Software Engineering 1999*.

[14] H. Gomaa et al., “A Prototype Domain Modeling Environment for Reusable Software Architectures”, *Int. Conf. on Software Reuse*, Rio de Janeiro, November 1994, 74-83.

[15] M. Griss, “Implementing Product-Line Features by Composing Component Aspects”, *First International Software Product-Line Conference*, Denver, Colorado., August 2000.

[16] I. Holland. “Specifying Reusable Components Using Contracts”, *ECOOP 1992*.

[17] D.L. Parnas, “On the Design and Development of Program Families”, *IEEE Transactions on Software Engineering*, March 1976.

[18] Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”, *ECOOP 1998*.

[19] M. VanHilst and D. Notkin, “Using C++ Templates to Implement Role-Based Designs”, *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.

[20] D.M. Weiss and C.T.R. Lai, *Software Product-Line Engineering*, Addison-Wesley, 1999.

JTS: Tools for Implementing Domain-Specific Languages

Don Batory, Bernie Lofaso, and Yannis Smaragdakis
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{batory, bernie, smaragd}@cs.utexas.edu

Abstract[1]

The Jakarta Tool Suite (JTS) aims to reduce substantially the cost of generator development by providing domain-independent tools for creating domain-specific languages and component-based generators called GenVoca generators. JTS is a set of precompiler-compiler tools for extending industrial programming languages (e.g., Java) with domain-specific constructs. JTS is itself a GenVoca generator, where precompilers for JTS-extended languages are constructed from components.

1 Introduction

Software generators are among the most effective methods of achieving software reuse. Generators reduce maintenance costs, produce more evolvable software, and provide significant increases in software productivity [Deu97, Die97, Kie96]. From a technical standpoint, generators are compilers for *domain-specific languages* (*DSLs*) or general-purpose programming languages with domain-specific extensions [Cor90, Sma97]. Such languages express fundamental abstractions of a domain using high-level programming constructs. The P2 data structure generator is an example: P2 extended the C language with cursor and container data types [Bat93-94]. This allowed P2 users to program in data-structure-specific abstractions, which resulted in substantial improvements in productivity, program clarity, and performance.

Implementing a domain-specific language as an extension of an existing programming language (called a *host language*) has several advantages. First, we can leverage off existing functionality and not have to re-implement common language constructs. Second, the extensions themselves only need to be transformed to the point where they are expressible in the host language. Third, existing infrastructure (e.g., development and debugging environments) can be reused. All these factors result into lower implementation costs for language developers and decreased transition and education costs for users.

Nevertheless, adding domain-specific constructs to a general programming language presents severe technical difficulties. Programming languages are generally not designed to be extensible, and the ones that are (e.g., Lisp and a variety of other functional languages) have not gained wide acceptance. Addressing the needs of the industry (where C, C++, and Java prevail) is paramount for promulgating generator technology. Our interest in DSLs comes from our work in the design and construction of component-based generators, called *GenVoca* generators [Bat92-97]. Target applications are specified as compositions of reusable components; GenVoca generators convert these compositions into source code. From our experience, there is a serious lack of tools to simplify the construction of these generators. We estimate that over 60% of the effort in building a GenVoca generator involves the creation of a largely domain-independent programming infrastructure (e.g., component specification languages, component composition languages, etc.).

The *Jakarta Tool Suite (JTS)* is aimed at providing this common infrastructure: it is a set of domain-independent tools for extending industrial programming languages with domain-specific constructs. JTS is designed specifically for creating DSLs and GenVoca generators. JTS consists of two tools: Jak and Bali. The *Jak* language is an extensible superset of Java that supports meta-programming (i.e., features that allow Java programs to write other Java programs). *Bali* is a tool for composing grammars. JTS is itself a GenVoca generator. Languages and language extensions are encapsulated as reusable components. A *JTS component* consists of a Bali grammar file

[1] This work was supported in part by Microsoft, Schlumberger, the University of Texas Applied Research Labs, and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

(which defines the syntax of a language or extension) and a set of Jak files (that define the semantics of the extension as syntactic transformations). Different combinations of these components yield different language variants. Bali and Jak work cooperatively to automatically convert a composition of components that defines a language variant into a preprocessor for that variant.

The implementation of JTS is bootstrapped: JTS is written in Jak and Jak is also the first customized language that has been produced by JTS. (That is, new extensions are written in Jak (Java); the Jak preprocessor is then extended by this new component so that it can be used to write other extensions, and so on). In the following sections, we review the current features of Jak and Bali and explain the strategy behind their implementation. Afterwards, we explain the novelty of JTS and differentiate JTS from other language specification and construction tools.

2 The Jak Language

Jak is an open, extensible superset of Java. It extends Java with support for meta-programming (i.e., features that enable Java programs to write other Java programs). In the following sections, we explain two key features of Jak —namely, AST constructors and Generation Scoping — that distinguish it from Java. Both have been implemented as JTS components and are examples of the kinds of language extensions that JTS is capable of expressing.

2.1 AST Constructors

JTS internally represents programs and code fragments as two kinds of trees. A *surface syntax tree (SST)* is a parse tree of a code fragment (as defined by some grammar). An *abstract syntax tree (AST)* is a semantically-checked SST that has been annotated with type declarations and references to the symbol table. An SST is converted into an AST by invoking the **`typecheck()`** method on the root of the tree. In this section, we present the (surface syntax) tree constructors and composition methods in Jak.

A *tree constructor* is a code-template operator, analogous to the Lisp **`quote`** construct. It converts a code fragment into an SST; the value of a constructor is a pointer to the root. The expression constructor **`exp{ … }exp`**, for example, encloses a syntactically correct Jak expression. When the constructor is evaluated, an SST for that expression is created, and the root of that tree is the result. Similarly, **`stm{ … }stm`** is the corresponding constructor for Jak statements. SSTs can be unparsed (into text) using the **`print()`** method:

```
AST_Exp x = exp{ 7 + z*8 }exp;
AST_Stm s = stm{ foo(3);
                 if (y<4) return r; }stm;

x.print( );    // outputs “7 + z*8”
s.print( );    // outputs “foo(3);
               //    if (y<4) return r;”
```

There are presently 17 different tree constructors in Jak, the most commonly used are listed in Table 1.

Code fragments are composed using *escapes*, the counterpart to the Lisp **`comma`** (unquote) construct. The example below shows a statement constructor with an escape **`$stm(body)`**. When the constructor is evaluated, the SST of **`body`** is substituted in the position at which its escape clause appears.

```
AST_Stm body = stm{ if(i>40) foo(i); }stm;
AST_Stm loop = stm{ for(i=1; i<10; i++) {
                         $stm(body); } }stm;

loop.print();
       // outputs “for (i=1; i<10;i++)
       // { if (i > 40) foo(i); }”
```

Unlike Lisp and Scheme which have only a single constructor operator (e.g., backquote/comma), multiple constructors in syntactically rich languages are common (e.g., [Wei93], [Chi96]). The main reason has to do with the ease of parsing code fragments. We avoided the complications described in [Wei93] by making explicit the type of SST that is returned by a tree constructor. The result is a slightly more complicated but robust system.

Constructor	Escape	Class	AST Representation Of
`exp{ … }exp`	**`$exp( … )`**	**`AST_Exp`**	expression
`stm{ … }stm`	**`$stm( … )`**	**`AST_Stmt`**	list of statements
`mth{ … }mth`	**`$mth( … )`**	**`AST_FieldDecl`**	list of data member and method declarations
`cls{ … }cls`	**`$cls( … )`**	**`AST_Class`**	list of class and interface declarations
`id{ … }id`	**`$id( … )`**	**`AST_QualifiedName`**	qualified name

Table 1: AST Constructors and Escapes

Although the tree constructors of Table 1 are presently specific to Jak, this will not always be the case. Tree constructors can be added for other languages, such as CORBA IDL, embedded SQL, (subsets of) C and C++, so that IDL code, embedded SQL code, etc. can be generated.

2.2 Generation Scoping

Tree constructors and escapes are not sufficient for code generators (meta-programs); there must also be a mechanism to solve the *variable binding* or *inadvertent capture* problem [Koh86], which arises when independently-written code fragments are composed. Consider the following parameterized macro that defines a variable **temp** and initializes it to be twice the value of parameter **x**:

```
macro(x) { int temp = 2*x; ... }
```

Now consider application code that defines a variable, also called **temp**, and that invokes **macro(temp)**:

```
int temp = 5;
macro(temp);
```

The code that is produced on expansion is incorrect:

```
int temp = 5;
{ int temp = 2*temp;  ... }    // wrong
```

The inner **temp** variable was to be initialized using the outer **temp** variable; instead the uninitialized inner **temp** variable is used to initialize itself! The problem is that the **temp** identifiers are not sufficient to disambiguate the variables that they reference.

Hygienic, lexically-scoped macros (HLSM) were designed to solve this problem. HLSM relies on a "painting" algorithm that ensures identifiers are bound to the correct variables [Ree91]. Often, HLSM is implemented as a preprocessing step that mangles variable names to ensure their uniqueness:

```
int temp_0 = 5;
{ int temp_1 = 2*temp_0; ... }  // right
```

HLSM's applicability is limited to *macros* (pattern-based source code transformations). Since JTS supports programmatic (as opposed to macro or pattern-based) tree construction, we devised *Generation Scoping (GS)*, an adaptation and generalization of HLSM that is suited for JTS. We originally developed GS for Microsoft's Intentional Programming (IP) system [Sim95], and used it to develop the DiSTiL generator, an IP-version of the P2 generator [Sma96-97]. The IP implementation of GS used handles to symbol table entries to represent variable references (see also [Tah97]). Since JTS produces domain-specific preprocessors, we chose an alternative implementation that mangles identifiers. In the following sections, we review its features.

2.2.1 GS Environments

A GS *environment* is a list of identifiers (i.e., class or interface names, data member or method names, etc.) that are local to a set of related code fragments. To ensure there is no inadvertent capture, local identifiers are mangled. Associated with each environment instance is a unique *mangle number*, an integer that is attached to an identifier to make it unique. For example, if an environment's mangle number is **005** and identifier **i** is to be mangled, identifier **i_005** is produced.

Environments are associated with classes; environment instances are associated with objects. Class **foo** below defines an **environment** with identifiers **i** and **j**. Each **foo** instance creates an environment containing identifiers **i** and **j**. Different **foo** instances represent distinct environment instances. Whenever a tree constructor is evaluated by a **foo** object, it does so in the context of that object's environment. Thus, if **x** and **y** are distinct **foo** instances, and **x.bar()** and **y.bar()** return code fragments, the returned fragments will be isomorphic in structure, but will have different names for **i** and **j**.

```
class foo {
   environment i, j;   // ids to mangle
   AST_Exp bar() { return exp{ i+j }exp; }
}

foo x = new foo();// assume mangle# is 000
foo y = new foo();// assume mangle# is 001

x.bar().print(); // yields “i_000+j_000”
y.bar().print(); // yields “i_001+j_001”
```

With the above capabilities, the variable binding problem presented earlier is easily avoided. One defines a class (**macroExample**) with an **environment** that contains the **temp** identifier. A method of this class (**macroCode**) uses a tree constructor to manufacture the body of the "macro". The **temp** variable that is defined internally to that tree is given a unique name via mangling, so inadvertent capture can not arise.

```
class macroExample {
   environment temp;

   AST_Stmt macroCode(AST_QualifiedName n)
   {   return stm{ int temp = 2*$id(n);
                 ... }stm;
   }
}
```

Since identifiers in an environment need to be explicitly designated, the JTS version of generation scoping is not fully automatic.[2] Associating environments with objects does, however, represent an improvement compared to the explicit creation of unique identifiers (as with Lisp's `gensym` [Gra96]) and the manual substitution of mangled names (via explicit escapes) into generated code fragments. Identifiers are now encapsulated and can be treated as a group. Additionally, these groups can be arranged in complex configurations, as we will see next.

2.2.2 GS Environment Hierarchies

Environment instances can be organized hierarchically to emulate scopes in the name space of generated programs. As expected, identifiers of parent environments are visible in child environments and identifiers that are declared in a child environment hide identifiers in parent environments with the same name. Parent linkages among environments are made at meta-program run-time using the `environment parent` declaration. The example below shows that instances of class `baz` make their environments children of environments of `foo` objects. Note that a tree constructor for the expression `“i + k”` produces `“i_000 + k_002”` because identifier `i` is mangled by the `foo` environment while `k` is mangled by the `baz` environment:

```
class baz {
   environment k;
   baz( foo z ) { environment parent z; }

   AST_Exp biff() {return exp{ i+k }exp; }
}
```

```
baz r = new baz(x);  // x has mangle # 000
                     // r has mangle # 002

r.biff().print();  // yields “i_000+k_002”
```

More generally, generation scoping allows environment instances to be arranged in directed acyclic graphs. This permits the visibility of identifiers from multiple parent environments, which is indispensable when building GenVoca generators. Detailed examples of generation scoping are presented in [Sma96].

2.3 Tree Traversals

Jak provides a Java package of classes for searching and editing trees using objects of type `Ast_Cursor`. Methods that can be performed on cursors are listed in Table 2.[3] In the code fragment below, a cursor `c` is used to examine every node of a tree and subtrees that define interface declarations are deleted.

```
Ast_Cursor c  = new Ast_Cursor();
Ast_Node Root = // root of AST to search

for(c.First(root); c.More(); c.PlusPlus())
   if (c.node instanceOf Ast_Interface)
       c.Delete();
```

2.4 Jak Extensibility

Representing programs internally as parse trees offers a powerful form of language extensibility. This principle has been widely explored in the Lisp community and various syntax tree formats are commonly used in transformations systems (e.g., Microsoft's IP [Sim95], Open C++ [Chi96]). New kinds of tree nodes can have domain-spe-

Cursor Operation	Meaning
`First(r)`	position cursor on root (`r`) of tree
`More()`	true if more nodes to examine in tree
`PlusPlus()`	advance cursor to next node of tree
`Sibling()`	skip the search of subtrees of current node
`Parent()`	reposition to parent of current subtree
`Delete()`	delete subtree rooted at current node
`Replace(x)`	replace the current node with `x`
`AddAfter(x)`	add tree `x` after the current node
`AddBefore(x)`	add tree `x` before the current node
`print()`	unparse the tree rooted at the current node

Table 2: Operations on Tree Cursors

[2] The IP version of GS automatically enters identifiers into environments as tree constructors are evaluated. The JTS version reflects a design that was used in the P2 generator, where manual declaration of identifiers was used.

[3] Tree editing methods guarantee syntactic correctness; however, they cannot guarantee semantic correctness.

cific semantics and transform, at reduction time, to a host language implementation (or, more accurately, a tree that defines this implementation). This approach is called *intention-based* programming[4] [Sim95]. For example, the P2 language extended the C language with cursor and container data types and operators. Tree nodes that represented these types and operations were intentions. At reduction time, a P2 program was converted into a pure C program by replacing cursor and container nodes with trees that defined their concrete C implementations.

JTS follows this approach (see Figure 1). A domain-specific program is converted into an AST by a lexical analyzer (lexer) and parser. The AST is then manipulated into another tree by a reduction program, and the resulting tree is unparsed into a pure host-language program (currently a Java program). Note that the reduction program itself is written in Jak, because Jak is specifically designed for tree creation and manipulation.

2.5 Perspective

Jak is an integration of a popular programming language (Java), with meta-programming concepts (tree constructors and generation scoping), and intention-based programming. The structure of Jak provides the basis for an inherently open precompiler. In the following sections, we answer the following questions:

- How are lexers and parsers produced by JTS?
- How is the reduction program produced by JTS?
- How is GenVoca related to JTS?

3 Bali: A GenVoca Generator of DSL Precompilers

Bali is the second tool of JTS. There are two aspects to Bali. First, it is a tool for writing precompilers for domain-specific languages. In this respect, Bali looks similar to other DSL-specification tools: the syntax of a DSL or language extension is specified using an annotated, extended BNF grammar. Second, Bali is a GenVoca generator. DSLs and their precompilers are specified as a composition of components; evolution of a DSL (e.g., adding and removing features) is accomplished by adding and removing components.

To show that Bali is a GenVoca generator, we will examine one of its most important applications: Jak itself. Jak is a preprocessor implemented as an extended version of Java using Bali. The reasoning behind this design decision is simple. Jak is really not a single language, but a family of related languages. There will be variants of Jak with/without generation scoping, variants with/without tree constructors, variants with/without CORBA IDL extensions, and so on. This a classical example of the *library scalability problem* [Bat93, Big94]: there are *n* features and often more than *n*! valid combinations (because composition order matters and feature replication is possible [Bat92]). It isn't possible or practical to build all combinations by hand. Instead, the specific instances that are needed can be generated. The JTS library presently includes components for the Java language, tree constructors, generation scoping, and domain-specific generators (e.g., P3 [Bat98]). Compositions of these components define a particular variant of Jak.

3.1 Bali as a DSL Compiler Tool

Bali transforms a Bali grammar into a preprocessor. A Bali grammar is BNF with regular-expression repetitions. For example, two Bali productions are shown below: one defines **StatementList** as a sequence of one or more **Statements**, while **ArgumentList** is defined as a sequence of one or more **Arguments** separated by commas. The use of repetitions simplifies grammar specifications [Wil93, Rea90a] and allows an efficient internal representation as a list of trees.

```
StatementList : ( Statement )+ ;
ArgumentList : Argument ( ‘,’ Argument )*;
```

Bali productions are annotated by the class of objects that is to be instantiated when the production is recognized. For example, consider the Bali specification of the Jak **SelectStmt** rule:

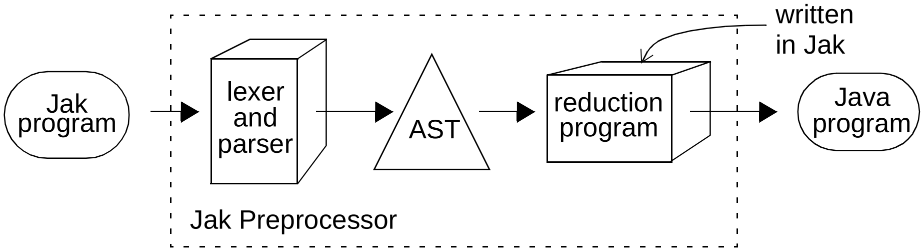


Figure 1: Data Flow in Jak

[4] Although the term is new, the idea is quite old. Lisp macros were powerful enough to express useful extensions to the language. They have been routinely used to encode new constructs in terms of core language constructs. We prefer, however, to use the term “macro” exclusively for pattern-based program transformations.

```
SelectStmt
 : IF ‘(’ Expression ‘)’ Statement ::IfStm
 | SWITCH ‘(’ Expression ‘)’ Block ::SwStm
 ;
```

When a parser recognizes an “if” statement (i.e., an **IF** token, followed by ‘**(**‘, **Expression**, ‘**)**’, and **Statement**), an object of class **IfStm** is created. Similarly, when the pattern defining a “switch” statement (a **SWITCH** token followed by ‘**(**‘, **Expression**, ‘**)**’, and **Block**) is recognized, an object of class **SwStm** is created. As a program is parsed, the parser instantiates the classes that annotate productions, and links these objects together to produce the SST of that program.

For each production, Bali infers (among other things) the constructors for tree nodes. Each parameter of a constructor corresponds to a token or nonterminal of a pattern.[5] For example, the constructor of the **IfStm** class has the following signature:

```
IfStm( Token iftok, Token lp,
 Expression exp, Token rp, Statement stm )
```

Methods for editing and unparsing nodes are additionally generated.

Bali also deduces an inheritance hierarchy of tree node classes. Consider Figure 2a which shows rules **Rule1** and **Rule2**. When an instance of **Rule1** is parsed, it may be an instance of **pattern1** (an object of class **C1**), or an instance of **Rule2** (an object of class **Rule2**). Similarly, an instance of **Rule2** is either an instance of **pattern2** (an object of **C2**) or an instance of **pattern3** (an object of **C3**). From this information, the inheritance hierarchy of Figure 2b is constructed: classes **C1** and **Rule2** are subclasses of **Rule1**, and **C2** and **C3** are subclasses of **Rule2**.

A Bali grammar specification is a streamlined document. It is a list of the lexical patterns that define the tokens of the grammar followed by a list of annotated productions that define the grammar itself. A Bali grammar for an elementary integer calculator is shown in Figure 3. To give readers an idea of the size of other grammars, the Jak grammar uses 160 tokens, 270 productions, defines 290 classes in 750 lines. The “meta” grammar for Bali grammars uses 20 tokens, 20 productions, defines 37 classes in 100 lines.

```
                // Lexeme definitions

     "print"  PRINT
     "+"      PLUS
     "-"      MINUS
     "("      LPAREN
     ")"      RPAREN
     "[0-9]*" INTEGER

     %%      // production definitions
             // start rule is Action

     Action  : PRINT Expr       :: Print
             ;

     Expr    : Expr PLUS Expr   :: Plus
             | Expr MINUS Expr  :: Minus
             | MINUS Expr       :: UnaryMinus
             | LPAREN Expr RPAREN :: Paren
             | INTEGER          :: Integer
             ;
```

Figure 3: A Bali Grammar for an Integer Calculator

Bali generates the following from a grammar specification:

- *A lexical analyzer.* We are using **Jlex**, a version of **lex** written in Java [Lof96].
- *A parser.* We are using **JavaCup**, a version of **yacc** written in Java [Hud96].
- *Inheritance hierarchies of tree node classes, with constructor, editing, and unparsing methods.*

There are obviously many methods that cannot be generated by Bali, including type checking, reduction, and optimization methods. Such methods are node-specific; we hand-code these methods and encapsulate them as subclasses of Bali-generated classes. (The reason for this will become clear in Section 3.2). Figure 4 shows the inheritance hierarchy of a Bali-generated precompiler. **AstNode** is the root of all node hierarchies; it is a hand-written class

(a)

```
Rule1 : pattern1   :: C1
      | Rule2
      ;

Rule2 : pattern2   :: C2
      | pattern3   :: C3
      ;
```

(b)

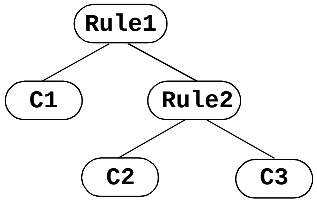


Figure 2: Grammar Inheritance Hierarchies

[5] The tokens need not be saved. However, Bali-produced precompilers presently save all white space — including comments — with tokens. In this way, JTS-produced tools that transform domain-specific programs will retain embedded comments. This is useful when debugging programs that have a mixture of generated and hand-written code, and is a necessary feature if transformed programs will subsequently be maintained by hand [Tok95].

in the JTS **kernel** package. Its immediate subclasses are the hierarchy of subclasses generated from a Bali grammar. The terminal classes of this hierarchy are hand-coded and define the type checking, reduction, and optimization methods for individual nodes. It is these terminal classes that are instantiated during the compilation of a domain-specific program.

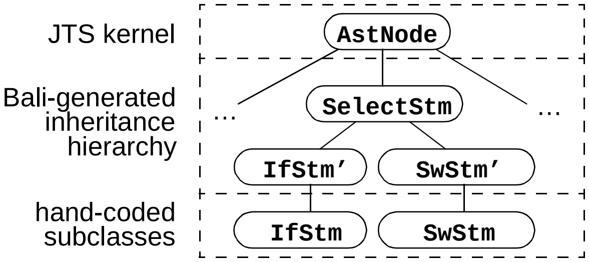


Figure 4: A Bali-Generated Class Hierarchy

3.2 Bali as a GenVoca Generator

GenVoca is a scalable model of component-based software construction [Bat92-97b]. The central idea is that software domains are characterized by a finite set of fundamental abstractions. By standardizing the programming interfaces to these abstractions, components can encapsulate reusable algorithms of a domain by exporting and importing standardized interfaces. A target system is specified by a composition of components called a *type equation*. Elementary compositions of components can be visualized as a stack of layers. Figure 5a depicts a system **S** where component **Z** sits atop **Y** which sits atop **X**. Its type equation is **S = Z[Y[X]]**.

GenVoca generators have been created for widely disparate domains. Interestingly, most have been written in the C language, and only two have been written in OO languages [Sin93-96, Van96]. A problem that we have faced for years but only very recently have been able to answer is: What is the relationship between GenVoca components and OO classes? The key lies in the relationship of layering and inheritance.

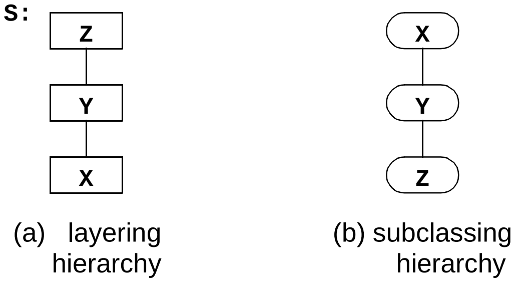


Figure 5: Layering and Subclassing Hierarchies

A common phenomenon in layered systems is *operation propagation* [Bat97b]: operations of lower layers are exported through the top of a system. In Figure 5a, suppose operation **g()** of layer **X** is to be exported by system **S**. This means that **g()** must be propagated through layers **Y** and **Z** (or in general, whatever layers are stacked above **X**). When an operation of **S** is called (such as **g()**), the corresponding operation of layer **Z** is invoked, which might call methods of layer **Y**, which further might call methods of **X**.

Now, suppose every component encapsulates a single class. To account for operation propagation *and* the processing of operations in layered systems, the subclassing hierarchy of Figure 6b comes to mind. Operation **g()** of class **X** is propagated to classes **Y** and **Z** by inheritance. Invoking an operation of **S** (such as **g()**) invokes the corresponding operation of class **Z**, which might call methods

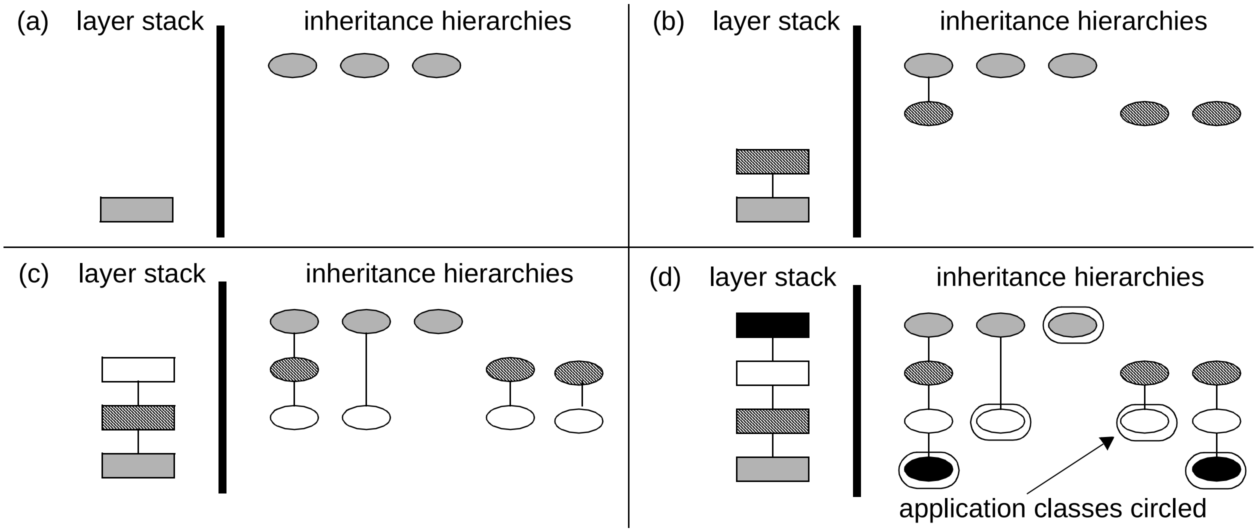


Figure 6: Component Composition and Inheritance Hierarchies

of class **Y**, which might further call methods of **X**. From this perspective, *inheritance hierarchies are inverted layer hierarchies*.[6]

The relationship of GenVoca components and OO classes can now be seen. *GenVoca components encapsulate suites of interrelated classes.* Figure 6a shows a (terminal) shaded layer that encapsulates three classes. Figure 6b shows a striped layer that also encapsulates three classes; when it is stacked on top of the shaded layer, one class becomes a subclass of the left-most shaded class, while the others begin new inheritance hierarchies. Figure 6c shows a white layer to encapsulate four classes. When stacked upon the striped layer, each of these classes becomes a subclass of an existing class. Lastly, Figure 6d shows the effect of adding a black layer, which encapsulates two classes. The application (which is defined by the resulting layer stack) instantiates the most refined classes (i.e., the terminal classes) of these hierarchies. These classes are circled in Figure 6d; the non-terminal classes represent intermediate derivations of the terminal classes. *Thus, when GenVoca components are composed, a forest of inheritance hierarchies is created. Adding a new component (stacking a new layer) causes the forest to get progressively broader and deeper* [Sma98].

The connection of these ideas to Bali and the Jak language is straightforward. A *JTS component* has two parts: The first is a Bali grammar file (which contains the lexical tokens and grammar rules that define the syntax of the host language or language extension). The second is a GenVoca component: a collection of multiple hand-coded classes that encapsulate the reduction, etc. methods for each tree node defined in that grammar file. These classes define the semantics of an extension. There are JTS components for Java (**Java**), SST constructors and explicit escapes (**SST**), generation scoping (**GScope**), and data structures (**P3** [Bat98]), among others.[7] The Jak language and precompiler is defined by a composition of these components, i.e., the type equation **Jak = P3[GScope[SST[Java]]]**.

The syntax of a composition is defined by taking the union of the sets of production rules in each JTS component grammar. The semantics of a composition is defined by composing the corresponding GenVoca components, as described previously. Figure 7 depicts the class hierarchy of the Jak precompiler. **AstNode** belongs to the JTS kernel, and is the root of all inheritance hierarchies that Bali generates. Using the composition grammar file (the union of the grammar files for the **Java**, **SST**, **GScope**, and **P3** components), Bali generates a hierarchy of classes that contain tree node constructors, unparsing, and editing methods. Each JTS component then grafts onto this hierarchy its hand-coded classes. These define the reduction, optimization, and type-checking methods of tree nodes by refining existing classes, just as in Figures 4 and 6. The terminal classes of this hierarchy are those that are instantiated by the generated preprocessor.

It is worth noting that Figure 7 is not drawn to scale. Jak consists of over 300 classes. The average number of classes that a language-extension component adds to an existing hierarchy ranges from 10 to 40. In terms of the number of classes a GenVoca component encapsulates, Bali components are clearly the largest we've ever encountered. However the simplicity and economy of specifying Jak using type equations is enormous: to build the Jak precompiler, all that users have to provide to Bali is the equation **Jak = P3[GScope[SST[Java]]]**, and Bali does the rest. To compose all these classes by hand (as would be required by Java) would be very slow, extremely tedious, and error prone. This is (another) good example why programming with reusable components (and hence at higher-levels of abstraction) offers big productivity gains. Additionally, the scalability advantages of GenVoca can easily be obtained: when new extension

[6] Note that the converse is not true; there are layer hierarchies that are not inheritance hierarchies. Inheritance hierarchies arise whenever layer hierarchies refine a single abstraction (e.g., classes **X**, **Y**, and **Z** are different implementations of the same concept). When layers implement different abstractions, class composition relies solely on parameterization and does not involve inheritance.

[7] Presently, Bali supports a single realm of components (**J**) that define and extend the Java language. Using the standard notation for realm definitions, **J = {Java, SST[J], GScope[J], P3[J],… }**.

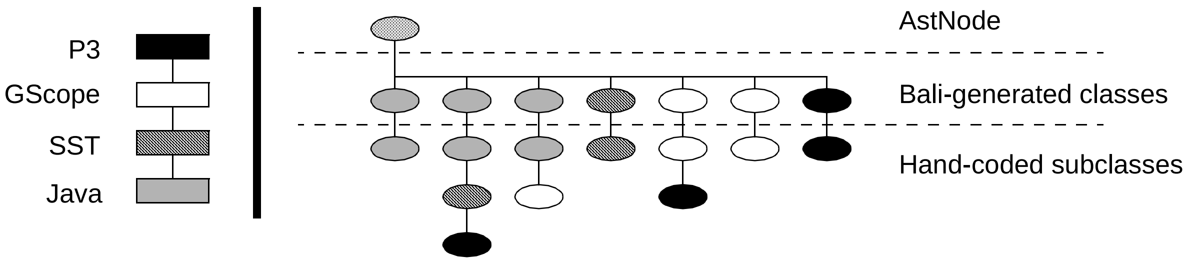


Figure 7: The Jak Inheritance Hierarchy

mechanisms or new base languages are specified as components, a subset of them can be selected and Bali will automatically compose a preprocessor for the desired language variant.

3.3 Perspective

The primary goal of JTS is to provide tool support for building GenVoca generators. Initially, it was unclear to us how language extensions could be encapsulated or composed; we feared that we would invent an ad hoc technology for defining and specifying components that was foreign to GenVoca. (This would then put a significant burden upon us to demonstrate the generality of this new model and its connection with other work, let alone how to address the odd situation of using a different component model to implement a more general model). Thus, realizing the connection between layering and inheritance hierarchies was a watershed event. It told us that JTS (or more specifically the central tool of JTS — Bali) was yet another example of GenVoca. Our focus immediately shifted away from an ad hoc implementation of Bali to one that would exhibit a principled and clean design.

4 Related Work

Meta-programming and syntactic transformations have been areas of active research for several decades. Since the volume of related work is enormous we will be selective in our presentation and only discuss approaches that are particularly close to JTS.

As should be evident from the previous sections, JTS is only concerned with the front-end of what is traditionally termed a *transformation system* (e.g., Draco [Nei89], Refine [Rea90b]). JTS mainly deals with parsing and the mechanics of syntactic transformation. Any sophistication of the transformation process (e.g., algebraic rewrites) will have to be provided by JTS client programs (e.g., the P3 generator [Bat98]).

Part of the novelty of JTS is that basic ideas of meta-programming and precompiler-compiler tools have transported to a "modern" and syntax-rich language platform (i.e., Java). The intricacies of our task are demonstrated, for instance, by the large variety of AST constructors discussed in Section 2.1 (compared to a single "quoting" operator for languages like Lisp).

Another contribution of JTS is in the way it achieves language extensibility: it does so through the prescripts of an architectural model (GenVoca): language extensions are encapsulated as components and languages and their preprocessors are assembled from these components.

It is instructive to compare this approach to that of Dialect [Rea90a]. Dialect is the front-end of the Refine transformation system and is in many ways analogous to the part of Bali described in Section 3.1. One of the biggest differences is in the way separate language extensions can be composed. By analogy to object-oriented programming, Dialect introduces the notion of *grammar inheritance*: a grammar (e.g., one defining a language extension) could "inherit" from another grammar (e.g., a base language). The resulting grammar is defined by taking the union of all productions contained in the two grammars — just like in JTS. An important difference, however, is that, unlike in Dialect, JTS grammars do not have to specify the grammar they are inheriting from. This is implicitly specified when grammar components are composed to form an entire language. The benefit is that a single JTS grammar component can be used to extend multiple base languages. Carrying the object-oriented programming analogy further, we could say that, instead of grammar inheritance, JTS allows *grammar mixins* (in the sense of OO *mixin classes* [Bra90]).

An interesting technical comment on comparing JTS with Dialect has to do with the way grammar rules are associated with inheritance between classes of AST nodes (see Section 3.1). Recall that JTS infers inheritance relationships from grammar rules. Conversely, Dialect requires that inheritance relationships be explicitly specified but infers grammar rules from them. The two approaches are semantically equivalent but we preferred having an explicit and compact representation of all grammar rules, as opposed to a mixed representation.

It is interesting to examine the relationship of JTS to *meta-object protocols (MOPs)*. The fundamental premise of a MOP is that class-specific extensions are themselves object-oriented in nature. Thus, they can be encapsulated in another class, called a *meta-class*. If a certain class *A* has meta-class *MA*, *A* is itself viewed as an object — an instance of *MA*. Methods of *MA* define extensions for all objects of *A*. For instance, methods of *MA* may define extension code for every construction of objects of class *A*, any assignment to such objects, or any method invocation on them.

Most MOPs are compositional: meta-classes contain code to be executed at a specified moment. There are, however MOPs where extensions are *transformational*: meta-classes contain code that transforms the source code of a class definition or use. The transformational MOP closest to JTS is Open C++ ([Chi95], [Chi96]). Open C++ encapsulates transformational extensions to C++ (i.e., syntax tree transforms, just like JTS) as meta-classes. Like JTS, Open C++ is implemented as a compiler that takes meta-class specifications as input and produces a preprocessor and compiler (packaged together) for extended C++ as output. Unlike JTS, however, no arbitrary syntactic extensions are allowed (the changes to the language syntax are of one of a few pre-determined forms). The reason has to do with the complicated syntax of C++ and the difficulty of adding more syntax rules to it. The complexity of arbitrary syntactic extensions in JTS is what led us to represent them as GenVoca components. Compared with Open C++, the elements of JTS have direct counterparts: Jak corresponds to the meta-programming part (language for transformational extensions), while Bali is the counterpart of the meta-object protocol. Now we can see the role of JTS extensions as GenVoca components. Just like Open C++ (or any MOP) represents class-specific extensions as (meta-)classes, JTS represents arbitrary syntactic extensions as GenVoca components (encapsulated suites of classes). The main purpose of JTS has been to facilitate adding extensions for building GenVoca components in Java. By making the extension mechanism structure similar to that of the intended applications, the JTS design exhibits the same kind of simplicity and self-containment as meta-object protocols for object-oriented languages.

Microsoft's Intentional Programming (IP) system is a visionary project that has similar goals to JTS [Sim95]. IP inherently supports language extensibility through syntactic rewrites. It is not, however, concerned with surface language syntax but operates directly on an abstract syntax tree representation of a program. Additionally, IP transformations have no inherent knowledge of the semantics of any particular programming language. The purpose of IP is to become a powerful enough transformation system so that entire languages can be expressed as collections of cooperating transformations. We considered using IP but did not do so for reasons that had to do both with its current state (under development) and with our funding requirement for public availability of our code. Additionally, we were interested in experimenting with an extensible language system implemented around ideas from object-oriented and component-based programming.

5 Conclusions

Future software development tools will feature the generation and transformation of OO programs. Such tools will automate certain aspects of software design methodologies that aim at reuse, namely automating OO design patterns and generating domain-specific software from declarative specifications. The *Jakarta Tool Suite (JTS)* is designed with these applications in mind. JTS is a careful integration of three different technologies — meta-programming, precompiler-compiler tools, and component-based generators. JTS is also aimed at a growing community of software developers — those that use Java — who will benefit most from such tools.

The novelties of JTS are its integration of technologies and that JTS is an example of the very software design paradigm it was intended to support — GenVoca. With appropriate language support, it *is* substantially easier to write generators. And with clearly written and documented examples, it should be much easier to transition component-based generator technology from academic environments to industry.

In this paper, we have reviewed the two tools that comprise JTS: Jak and Bali. Jak is a JTS-produced language that extends Java with meta-programming features (e.g., tree constructors, generation scoping). Bali is the generator that produced Jak through component assemblies. The first GenVoca generator that we have built using JTS is P3 [Bat98], a Jak-based version of the P2 data structures generator. P3 was developed in a fraction of the time that was needed for P2. Moreover, the source code of P3 is *substantially* more elegant, readable, and maintainable because JTS provides the appropriate language constructs for building such generators. Further work on JTS will extend Jak to have language support for component definitions and compositions.

JTS runs on **Unix (Solaris)**, and **Windows (95** and **NT**) platforms. A beta-release became available in February 1998. For current information, release announcements, and the latest technical reports, please check our web page **`http://www.cs.utexas.edu/users/schwartz`**.

6 References

[Bat92] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[Bat94] D. Batory, J. Thomas, and M. Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler", *ACM SIGSOFT* 1994.

[Bat97a] G. Jimenez-Perez and D. Batory, "Memory Simulators and Software Generators", *1997 Symposium on Software Reuse*.

[Bat97b] D. Batory, "Component Validation and Subjectivity in GenVoca Generators", *IEEE Trans. Software Engineering, February 1997*.

[Bat98] D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for Generators", *Int. Conference Software Reuse*, 1998.

[Big94] T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse", *International Conference on Software Reuse*, 1994.

[Bra90] G. Bracha and W. Cook, "Mixin-Based Inheritance", *ECOOP/OOPSLA 90*, 303-311.

[Chi95] S. Chiba, "A Metaobject Protocol for C++", *OOPSLA 1995*.

[Chi96] S. Chiba, “Open C++ Programmer's Guide for Version 2”, SPL-96-024, Xerox PARC, 1996.

[Cor90] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow, “TXL: A Rapid Prototyping System for Programming Language Dialects”, *Computer Languages*, Vol. 16#1, 1991, 97-101.

[Deu97] A. van Deursen and P. Klint, “Little Languages: Little Maintenance?”, *Proc. SIGPLAN Workshop on Domain-Specific Languages*, 1997.

[Die97] P. Dietz, C. Jervis, M. Kogan, and T. Weigert, “Automated Generation of Marshalling Code from High-Level Specifications”, RNSG Research, Motorola, Schaumburg, Illinois, 1997.

[Gra96] P. Graham, *ANSI Common Lisp*, Prentice Hall, 1996.

[Hud96] S.E. Hudson, “Cup Users Manual”, Graphics Visualization and Usability Center, Georgia Institute of Technology, March 1996.

[Kie96] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A.Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith and L. Walton, “A Software Engineering Experiment in Software Component Generation”, *ICSE*1996.

[Koh86] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, “Hygienic Macro Expansion”. In *SIGPLAN ‘86 ACM Conference on Lisp and Functional Programming*, 151-161.

[Lof96] B. Lofaso, “JLex Users Manual”, University of Texas Applied Research Laboratories, 1996.

[Nei89] J. Neighbors, “Draco: A Method for Engineering Reusable Software Components”. In *Software Reusability*, T.J. Biggerstaff and A. Perlis, eds, Addison-Wesley/ACM Press, 1989.

[Rea90a] Reasoning Systems, “Dialect User’s Guide”, Palo Alto, California, 1990.

[Rea90b] Reasoning Systems, “Refine 3.0 User’s Guide”, Palo Alto, California, 1990.

[Ree91] W. Clinger and J. Rees. “Macros that Work”. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991, 155-162.

[Sim95] C. Simonyi, “The Death of Computer Languages, the Birth of Intentional Programming”, *NATO Science Committee Conference, 1995.*

[Sin93] V. Singhal and D. Batory. “P++: A Language for Large-Scale Reusable Software Components”, *6th Annual Workshop on Software Reuse* (Owego, New York), Nov. 1993.

[Sin96] V.P. Singhal. “A Programming Language for Writing Domain-Specific Software System Generators”. Dept. of Computer Sciences, University of Texas at Austin, September 1996.

[Sma96] Y. Smaragdakis and D. Batory, “Scoping Constructs for Program Generators”. TR 96-37, Dept. of Computer Sciences, University of Texas at Austin, December 1996.

[Sma97] Y. Smaragdakis and D. Batory, “DiSTiL: a Transformation Library for Data Structures”, *USENIX Conf. on Domain-Specific Languages*, 1997.

[Sma98] Y. Smaragdakis and D. Batory, "Implementing Reusable Object-Oriented Components", *Int. Conference on Software Reuse* 1998.

[Tah97] W. Taha and T. Sheard, "Multi-Stage Programming with Explicit Annotations", *Partial Evaluation and Semantics Based Program Manipulation (PEPM‘97)*, June, 1997, Amsterdam.

[Tok95] L. Tokuda and D. Batory, "Automated Software Evolution via Design Pattern Transformations", *Symp. on Applied Corporate Computing*, Monterrey, Mexico, Oct. 1995.

[Wei93] D. Weise and R.Crew, “Programmable Syntax Macros”, *ACM SIGPLAN Notices* 28(6), 1993, 156-165.

[Wil93] D.S. Wile, “POPART: Producer of Parsers and Related Tools”, USC/ISI, November 1993.

[Van96] M. VanHilst and D. Notkin, “Using Role Components to Implement Collaboration-Based Designs”, *OOPSLA 1996*, 359-369.

Object-Oriented Frameworks and Product-Lines[1]

Don Batory[i], Rich Cardone[i], & Yannis Smaragdakis[ii]

Department of Computer Sciences, The University of Texas, Austin, Texas 78712 [i] *&*
College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332 [ii]
{batory, richcar}@cs.utexas.edu, yannis@cc.gatech.edu

Keywords: object-oriented frameworks, refinements, components, product-line architectures, GenVoca.

Abstract: Frameworks are a common object-oriented code-structuring technique that is used in application product-lines. A *framework* is a set of abstract classes that embody an abstract design; a *framework instance* is a set of concrete classes that subclass abstract classes to provide an executable subsystem. Frameworks are designed for reuse: abstract classes encapsulate common code and concrete classes encapsulate instance-specific code. Unfortunately, this delineation of reusable v.s. instance-specific code is problematic. Concrete classes of different framework instances can have much in common and there can be variations in abstract classes, all of which lead to unnecessary code replication. In this paper, we show how to overcome these limitations by decomposing frameworks and framework instances into primitive and reusable components. Doing so reduces code replication and creates a component-based product-line of frameworks and framework instances.

1 INTRODUCTION

A *product-line architecture (PLA)* is a design for a family of related applications. The motivation for PLAs is to simplify the design and maintenance of program families and to address the needs of highly customizable applications in an economical manner. Although the idea of product families is old (McIlroy, 1968; Parnas, 1976), it is an area of research that is only now gaining importance in software design (Bosch, 1998; DeBaud, 1999;

[1] This work was supported in part by Microsoft, Schlumberger, the University of Texas Applied Research Labs, and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

Gomaa et al., 1994; Cohen and Northrup, 1998; Batory 1998; Weiss and Lai, 1999; Czarnecki and Eisenecker 1999).

A *framework* is a collection of abstract classes that encapsulate common algorithms of a family of applications (Johnson and Foot 1998). Certain methods of these classes are left unspecified (and hence are "abstract") because they would expose details that would vary among particular, fully-executable implementations. Thus a framework is a "code template"—key methods and other details still need to be supplied, but all common code is present in abstract classes. A *framework instance* provides the missing details. It is a pairing of a concrete subclass with each abstract class of the framework to provide a complete implementation. These subclasses encapsulate implementations of the abstract methods, as well as other details (e.g., data members specific to a particular framework instance).

Frameworks often arise in PLA implementations. This is hardly unexpected: frameworks are appropriate for reusing software parts and specializing them in multiple ways for distinct applications. Members of a product-line can be created by refining framework classes with appropriate concrete implementations. Frameworks are a fundamental technique because of their simplicity and generality—from an implementation standpoint, frameworks are just a coordinated use of inheritance. Since inheritance is a fundamental mechanism in object-oriented languages, the applicability of the framework approach is wide.

The factoring of common algorithms into reusable, abstract classes greatly reduces the cost of software development when building a new product-line application (i.e., when creating a framework instance). Unfortunately, this delineation of reusable vs. instance-specific code has problems. Concrete classes of different framework instances can have much in common. This situation is typically addressed by either copying code (with predictable maintenance problems) or redeveloping concrete classes from scratch (which is costly). A different problem arises with abstract classes: they can have variations. Much of the code in abstract classes would be common across variations, but some parts would be radically different. Variability is typically handled by replicating the abstract classes of frameworks and hard-coding the changes into a new framework. Framework proliferation ensues, again incurring maintenance problems.

These problems are real. In a recent discussion with a member of IBM's SanFrancisco project,[2] these limitations of frameworks have become apparent. While code replication is not yet burdensome because SanFrancisco is still new, difficulties are anticipated in the future. We believe that these problems arise in other projects that use OO frameworks.

A simple way to state the above problem is that product lines with optional features will not be concisely expressed using frameworks. Such frameworks suffer from either “overfeaturing” (Codenie, De Hondt, Steyaert, and Vercammen 1997) —a lot of not-entirely-general functionality is part of the framework— or replication of the same code across many instances. In this paper we present a general technique to solve this problem. The solution is effected by *relaxing the boundary between a framework (the common part of the product line) and its instantiations (the product-specific part)*. More specifically our technique is based on assembling both the abstract classes of frameworks *and* the concrete classes of their instances from primitive and reusable components. We show that the level of abstraction that delineates abstract classes from concrete classes can be drawn in different ways, and by decomposing frameworks and their instances in terms of our components, variations in abstract and concrete classes can be handled without code replication. A particular framework or framework instance is created by a composition of components; variations in frameworks and their instances are expressed as different component compositions. Our approach can be used to express any framework, but requires more language support than plain frameworks—instead of regular inheritance, parameterized inheritance is needed. Nevertheless, this support is readily available in production languages like C++. An example in an extended version of the Java language is given to illustrate our ideas.

2 GENVOCA AND COLLABORATION-BASED DESIGNS

In this section we offer an overview of some design-level ideas that underlie our work. Many terms and concepts used in the rest of the paper are defined here.

Collaboration-Based Designs. It is well-known in object-oriented design that objects are encapsulated entities that are rarely self-sufficient. The semantics of an object is largely defined by its relationship with other objects. Object interdependencies can be expressed as collaborations. A

[2] IBM SanFrancisco is a Java-based set of components that allows developers to assemble server-side e-business applications from existing parts, rather than building applications from scratch.

collaboration is a set of objects and a protocol (i.e., a set of allowed behaviors) that determines how these objects interact. The part of an object that enforces the protocol of a collaboration is called the object's *role* in that collaboration (Reenskaug, et al, 1992; VanHilst and Notkin 1996; Smaragdakis and Batory 1998).

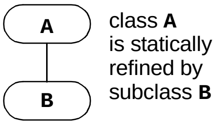
A *collaboration-based design* expresses an application as a composition of separately-definable collaborations. In this way, each object of an application represents a collection of roles describing actions on common data. Each collaboration, in turn, is a collection of roles that encapsulates relationships across its corresponding objects.

> **Example**. I (Batory) am an author of this paper. This relationship is defined by a collaboration of two objects, one in the role of author and another in the role of manuscript. I am also a parent—a collaboration of two or more objects, one in the role of parent and others in the role of children. I am also a car owner—a collaboration of two or more objects, one in the role of owner and others in the role of possession. And so on. I am a single object where relationships that I have with other objects are expressed through collaborations where I play a specific role in each collaboration.
>
> The collaborations mentioned above are generic; they are not specific to me but are general relationships that can be defined in isolation of each other. Furthermore, such collaborations are applicable in different contexts to different entities. For example, a corporation can own a car, and so too can a government entity. The relationship between owner and possession is the same in all cases, but the ownership roles are played by very different classes of objects (e.g., people, corporations, government). Symmetrically, a possession could be a car, dog, or building; the possession role can also be played by very different classes of objects. The same holds for roles in other collaborations.

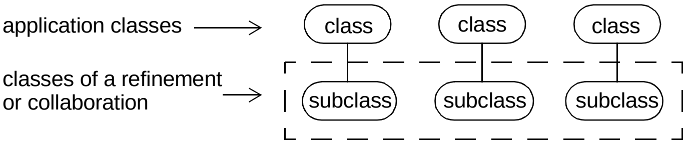
GenVoca. GenVoca is a design methodology for building architecturally-extensible software—i.e., software that is extensible via component addition and removal (Batory and O'Malley 1992). GenVoca is a scalable outgrowth of an old and practitioner-ignored methodology of program construction called *step-wise refinement*. GenVoca freshens this methodology by scaling refinements to a *component* or *layer* (i.e., multi-class-modularization) granularity, so that applications of enormous complexity can be expressed as a composition of a few refinements rather than hundreds or thousands of small refinements (c.f. (Partsch and Steinbruggen, 1983)).

Relationship. GenVoca product-lines and collaboration-based designs are intimately related: a GenVoca refinement is a collaboration. An application is constructed by composing reusable refinements (collaborations). Refinements can be composed dynamically at application run-time or statically at application compile-time. To address the problems of OO frameworks noted in the Introduction, we consider only statically composable refinements in this paper.

Consider how refinements are expressed statically in OO languages. A *static refinement* of an individual class adds new data members, new methods, and/or overrides existing methods. Such changes are expressed through subclassing: class **A** is refined by subclass **B**:



Both collaboration-based designs and GenVoca deal with *large-scale refinements*: such refinements involve the addition of new data members, new methods, overriding existing methods, etc. simultaneously to *several* classes:



The encapsulation of these “refining” subclasses in the above figure defines both a GenVoca *component* or *layer* and a collaboration. (Note that we are showing only subclassing relationships in this figure; there can be any number of “horizontal” interrelationships among individual subclasses).

> **Example**. Have you ever added a new feature to an existing OO application? If so, you discover that changes are rarely localized. Multiple classes of an application must be updated simultaneously and consistently for the feature to work properly. Similarly, if one subsequently removes that feature, all of its updates must be simultaneously removed from all affected classes. It is this collection of changes that we want to encapsulate as an application building block.

Each subclass of a layer encapsulates a role of a collaboration-based design. For a collaboration-based design to be “hooked” into an application, each role must be bound with an existing class of the application. We will see in Section 3 that layers can be expressed as *templates* and that such binding is accomplished via *template parameterization*. Thus, a layer defines a collaboration, while a layer instance additionally defines role/class bindings.

Compositions. When a layer (collaboration) is composed with other layers, a forest of subclassing (inheritance) hierarchies is created. As more layers are composed, the hierarchies become progressively broader and deeper. Figure 1 illustrates this concept: layer `L1` encapsulates three classes. Each of these classes root a subclassing hierarchy. Layer `L2` encapsulates three classes, two classes refine existing classes of `L1` while a third starts a new hierarchy. Layer `L3` also encapsulates three classes, two of which refine classes of `L1` and `L2`. Finally, layer `L4` encapsulates two classes, both of which refine existing classes.

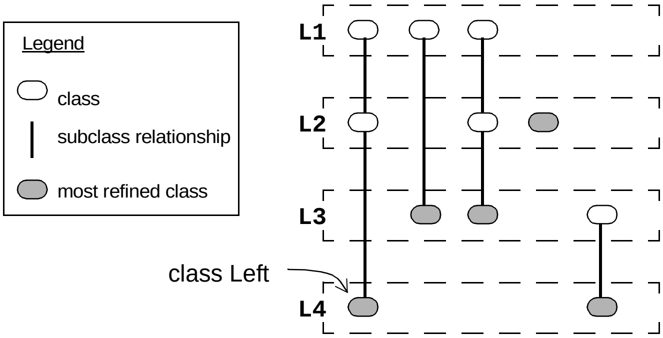


Figure 1 Creating Inheritance Hierarchies by Composing Layers

Each inheritance or refinement chain of Figure 1 represents a *derivation* of its terminal class. That is, each terminal class (shaded in Figure 1) is a product of its superclasses, where each superclass defines a role in some collaboration. In general, the classes that are instantiated by an application are the terminal classes, because these classes encode all the roles that are required of application objects. (For example, an object of class `Left` in Figure 1 plays three distinct roles which originate from collaborations `L1`, `L2`, and `L4`). The non-terminal (non-shaded) subclasses represent intermediate derivations of application classes. Thus, the composition of layers `L1` through `L4` yields five classes (i.e., those that are shaded in Figure 1); the unshaded classes represent the “intermediate” derivations of these shaded classes. If the resulting complexity of class hierarchies is a concern, preprocessors can be built to “accordion” (compact) inheritance chains so that only the terminal classes remain. The resulting classes would be *conceptually* layered, but not *physically* layered (Habermann, Flon, and Cooprider, 1976). In general, collaboration-based designs ultimately reduce complexity by shifting the design emphasis from small-scale components (individual classes) to large-scale ones (entire collaborations).

Product-Lines. A layer implements a *feature* (*aspect*, *capability*) that can be shared by many applications of a product-line. An application that supports a given set of features is defined by a composition of layers that implements those features. Thus, layers are the basic building blocks for families of applications. In general, *n* layers can be composed in excess of *n*! ways, because the order of composition matters and layer replication is possible.[3]

3 MIXIN-LAYERS

A *mixin-layer* is a template representation of a GenVoca refinement (Smaragdakis and Batory, 1998; Findler and Flatt, 1998). Templates are important in our methodology because they allow writing source-code components that can be used in multiple contexts. We will use *Jak*, a superset of Java that supports templates, to explain the basic technique (Batory, Lofaso, and Smaragdakis, 1998).[4] Mixin-layers are an interesting meld of parameterized inheritance, inner classes, and standardized naming conventions.

Mixins. A *mixin* is a class whose superclass is specified by a parameter (Bracha and Cook, 1990).[5] Below we define a mixin **`M`** whose superclass is defined by parameter **`S`**. **`AnyClass`** is a Java interface that all classes implement:

```
class M <AnyClass S> extends S { ... }
```

[3] So it is not uncommon that different applications of a product-line can be assembled by composing exactly the same layers in different orders (Batory and O'Malley, 1992; Hayden, 1998). Figure 1 provides an illustration: the order of **`L2`**-**`L4`** could be permuted, provided that **`L4`** is "below" **`L3`**.

[4] We will not offer a strict definition of the *Jak* language, but its diversions from Java are few and, we hope, mostly self-explanatory. The reader can think of the semantics of our parameterization mechanism (templates) as those resulting from straightforward textual substitution. We will not address the potential problems of an actual textual substitution implementation as these are orthogonal to the topic of this paper.

[5] C++ has a different meaning of "mixin" that is *not* equivalent to our use.

Mixins provide the capability of creating customized inheritance hierarchies when they are composed.

Nested Classes. Class declarations in Java can be nested inside other class declarations, and are inherited like data members and methods. Consider the following example:

```
class Parent { class Inner { ... } }
class Child extends Parent { }
```

`Child` is a subclass of **`Parent`**. Although no **`Child.Inner`** class is explicitly defined, such a class does exist as it is inherited from **`Parent`**. Nested classes emulate the encapsulation of multiple classes within a package, except this representation allows "packages" to appear as nodes in inheritance hierarchies.

Combining Ideas. A *mixin-layer* is an implementation of a collaboration. It is a mixin with nested classes, where each nested class corresponds to a role of a collaboration. A general form of a mixin-layer **`M`** is a *Jak* template that has *n+1* parameters: one parameter **`S`** that defines the superclass of **`M`**, plus *n* additional parameters {**r**$_1$ ...**r**$_n$} that define the specific classes the collaboration's role classes are to refine:

```
class M <AnyClass S, AnyClass r$_1$, ... AnyClass r$_n$>
extends S {
   class role$_1$ extends r$_1$ { ... }
   ...
   class role$_n$ extends r$_n$ { ... }
   ...
   // additional non-refining classes (if any)
}
```

When a domain is decomposed into primitive collaborations, experience has shown that different collaborations use the same names for roles, and classes that have the same role names refine each other when their collaborations are composed. While the above template for mixin-layer **`M`** is general, a much more compact form standardizes names of inner classes and eliminates role-class parameters to yield a template with a single parameter **`S`**, the mixin-layer's superclass:

```
class M <AnyClass S> extends S {
   class role$_1$ extends S.role$_1$ { ... }
   ...
   class role$_n$ extends S.role$_n$ { ... }
   ...
   // additional non-refining classes (if any)
}
```

This form of a source-code component is interesting because it enforces a clear structure. Mixin-layers are written in such a way that they are composable with each other. Yet, composing layers entails fixing the superclass of *all* classes inside a layer. Thus, layers are simple to use (e.g., they have a single parameter), but can affect large portions of a software application.

Collaboration Typing. Astute readers may have noticed that mixin-layer `M` should not have an unconstrained parameter; substituting an arbitrary class for parameter `S` is unlikely to work. A legal instantiation of `S` must satisfy constraints, e.g., it must have inner classes {$\mathtt{role}_1$ … $\mathtt{role}_n$}. This gives rise to the notion that collaborations (layers) are typed and so too are their parameters. Unfortunately, Java currently offers no support for type-checking nested classes. For instance, it is not possible to use an `implements` clause to express the requirement that a class should contain a nested class that supports a certain interface. Therefore, we limit ourselves to very simple properties that can be expressed in the Java type system. Namely, we use an empty interface `R`. Collaborations that “implement” `R` are said to be of type `R`; parameters of type `R` will be legally instantiated only by collaborations of type `R`. Thus the set of layers of Figure 1 can be represented as:

```
interface R { }                   // empty

class L1
   implements R { ... }           // mixin-layer L1
class L2 <R x> extends x
   implements R { ... }           // mixin-layer L2
class L3 <R x> extends x
   implements R { ... }           // mixin-layer L3
class L4 <R x> extends x
   implements R { ... }           // mixin-layer L4
```

Although typing collaborations in this manner goes a long way toward ensuring that parameters have legal instantiations, additional properties are needed to distinguish the case where components with identical interfaces have different semantics (Batory and Geraci, 1997; Smaragdakis and Batory, 1998). For the purposes of this paper (and without loss of generality), we make the simplifying assumption that typing is sufficient.

Compositions. Refinements are composed by instantiating one mixin-layer with another as its parameter. The two classes are then linked as a parent-child pair in an inheritance hierarchy. The final product of a collaboration composition is a class `Fig1` with the general form (expressed in Jak):

```
class Fig1 extends L4< L3< L2< L1 >>>                (1)
```

That is, **`L4`**, **`L3`**, …, **`L1`** are mixin-layers, “**`<...>`**” is the *Jak* operator for template instantiation, and **`Fig1`** is the name given to the class that is produced by this composition. (This particular composition corresponds to Figure 1). The classes of **`Fig1`** are referenced in the usual way, namely **`Fig1.role`**$_1$ defines the application class **`role`**$_1$, etc. Readers who are familiar with GenVoca will recognize such compositions as *type equations*, which has an alternative and more compact syntax:

```
Fig1 = L4< L3< L2< L1 >>>  // type equation of (1)   (2)
```

The space of all type equations corresponds to all applications that can be synthesized in this product-line.

4 LIMITATIONS OF OO FRAMEWORKS

A common case where frameworks prove to be too rigid is that of optional features. If a set of features are often but not always used, they cannot be encoded in the framework. (Otherwise, they will burden or render incorrect any framework instances not needing these features.) Thus, such features need to be encoded independently (i.e., replicated) in each framework instance that uses them. We will show in this section that using mixin-layers as building blocks for frameworks and their instances, we can encode an optional feature as a mixin-layer and include or exclude it at will from a specific composition.

Recall that a framework is a set of classes. For simplicity, our prior discussions assumed that all framework classes are abstract, but in general they need not be. Non-abstract classes could encapsulate a capability that is shared by (and can be optionally extended by) all framework instances. We will proceed under this more general setting. We also assume that mixin-layers have *no* variations (e.g., no optionally-selected algorithms) and that their collaborations are “monolithic”. Variations in product-line applications arise *only* from variations in compositions of mixin-layers. We will relax this assumption later.

To see the relationship between mixin-layers and frameworks, consider Figure 2a which replicates the inheritance hierarchies of Figure 1. Suppose we drew a line between layers **`L2`** and **`L3`**, where classes above the line define the classes of a framework. In Figure 2a, there would be four such classes {**`A`**$_1$, **`A`**$_2$, **`A`**$_3$, **`A`**$_4$}. Note that these classes correspond to the “most refined” classes of the refinement chains that lie above the line. The most refined classes that lie below the line define the concrete classes of a framework instance. In Figure 2a, there would be four such classes {C_1, C_2, C_3, C_5}. (Note that for this framework instance, A_4 need not be subclassed/refined). If we had a language preprocessor that would "accordion" (compact) refinement chains so that only the most refined classes remained, Figure 2b shows the result of this compaction. Readers will recognize Figure 2b as an encoding of a framework's classes and its instance classes.

Two points are worth noting. First, the classes of the framework of Figure 2 are defined by the type equation `F` = `L2<L1>`. An instance of this framework is any type equation whose innermost term is `F` (e.g., `Fig1` = `L4<L3<F>>`). *From this we can conclude that mixin-layers are building blocks of both frameworks and framework instances.*

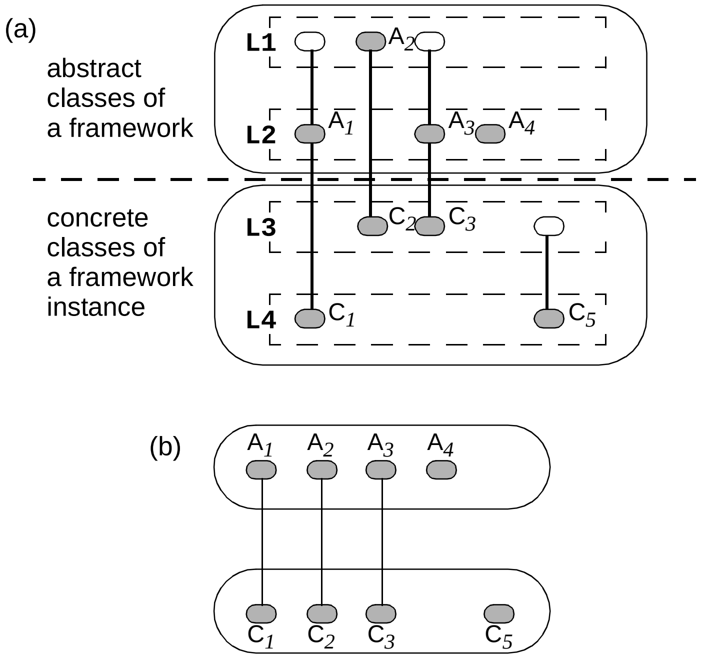

(a)
abstract classes of a framework
concrete classes of a framework instance
L1
L2
L3
L4
A1
A2
A3
A4
C1
C2
C3
C5
(b)
A1
A2
A3
A4
C1
C2
C3
C5

Figure 2: Refinement Hierarchies and Framework Instances

Second, we could have drawn our line in between any two adjacent layers and produced a different framework and one of its instances. (There is nothing special about the boundary between `L2` and `L3`). If the boundary is raised, the framework will become more general (as framework classes are simplified), but more code will need to be written for a framework instance. If the boundary is lowered, framework classes will encapsulate more features at an expense that the framework may be too specific (i.e., have too many features) to be used for a particular application.

But why partition along layer boundaries? Why not partition *across* layer boundaries? To see the answer, consider Figure 3 which defines the partitioning between framework and instance classes by crossing multiple layer boundaries. The framework of Figure 3 would consist of classes {$\mathtt{A}_1$, $\mathtt{A}_2$, $\mathtt{A}_3$}. The framework instance that is depicted would consist of concrete classes {$\mathtt{C}_1$, $\mathtt{C}_3$, $\mathtt{C}_4$, $\mathtt{C}_5$}, where $\mathtt{C}_3$ includes superclass $\mathtt{K}_3$, and $\mathtt{C}_5$ includes superclass $\mathtt{K}_5$.

Look carefully at what Figure 3 implies: *any* instance of the framework of Figure 3 *must* replicate classes $\mathtt{C}_3$ (which includes $\mathtt{K}_3$), $\mathtt{C}_4$, and $\mathtt{K}_5$. The reason is simple: all collaborations encapsulate the implementation of some primitive feature that is shared by many applications of a product line. If the classes of a framework implement only part of features `L2` and `L3` (which they do in this case), then any legal instance of this framework must supply the missing parts. Classes $\mathtt{C}_3$, $\mathtt{K}_3$, $\mathtt{C}_4$, and $\mathtt{K}_5$ are the missing parts and these parts do not vary (as we assumed at the beginning of this section). The framework of Figure 3 is a bad design because it forces the same code to be replicated in every framework instance. The only framework designs for which this isn't true are those that partition framework code from instance code along layer boundaries. Stated another way, the classes of a framework must fully implement an integral number of collaborations otherwise code replication in framework instances will occur.

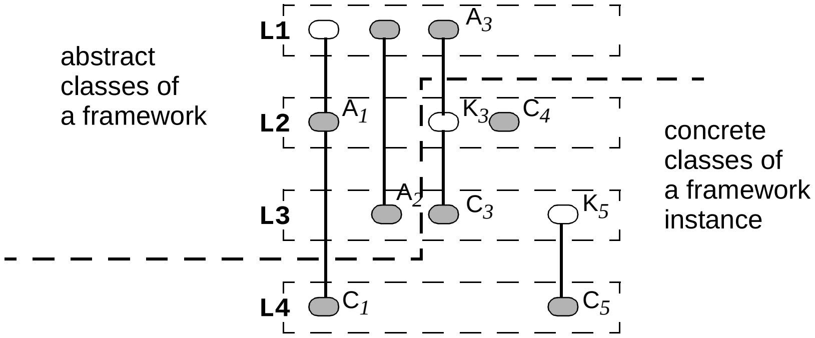

L1
A3
abstract classes of a framework
L2
A1
K3
C4
concrete classes of a framework instance
A2
L3
C3
K5
L4
C1
C5

Figure 3: Partitioning Against Layer Boundaries

From our experience, the above model of framework construction covers a majority of situations that are encountered in practice. That is, variations in frameworks and their instances can be explained as the composition of monolithic mixin-layers. Occasionally however, mixin-layers and their collaborations do have variations. If there are few variations, one could define a separate mixin-layer for each variant (see example of the next section). If there are a large number of variations, it is not difficult at all to create a generator of mixin-layers that will produce the desired variant of a collaboration from a high-level specification (Batory, Singhal, Sirkin, and Thomas, 1993). Doing so retains the building-block nature of mixin-layers (as we have been advocating) while forming a more compact encoding of the library of mixin-layers that must be maintained. By following either of these approaches, we can restructure the problem so that “good” framework designs can always be expressed as a partitioning *along* layer boundaries and not *across* layer boundaries.

Without loss of generality, let us assume “good” designs whose framework-instance partitioning corresponds to a layer boundary. We can now understand the problems of OO frameworks that we noted in the Introduction. Variations in framework classes arise whenever the type equation of a framework changes. Consider framework **`F`** whose type equation is **`F`** = **`L2<L1>`**. Any change to this equation—by swapping components (**`F1`** = **`L2<L0>`** for some new terminal component **`L0`**) or adding new components (**`F2`** = **`L2<L5<L0>>`** for some new nonterminal component **`L5`**)—will cause the classes of the framework to change.

Similarly, code repetition in multiple framework instances corresponds to the situation where the type equations of framework instances share the same framework subexpression and some (not necessarily all) remaining components. Consider equations **`Fig1 = L4<L3<F>>`** and **`Q = L4<L5<F>>`**. Both framework instances are distinct, but share the same framework subexpression (**`F`**) and component **`L4`**.

A way to avoid code replication problems in frameworks is to decompose the domain that a framework and its instances represent into a library of refinements/components. These are the components that should be given to application developers. They will choose which type equation (component composition) that they need to define their framework and/or framework instance. The boundary where an abstract class ends and concrete classes begin is left up to the developer. We would expect that the library of components that is distributed to be incomplete. That is, we would not expect the library to have enough components to construct a target framework instance (because we anticipate novel capabilities to be added by the instance). However, if there are sufficient components (or even just a few that can be reused), application developers will be further along their software development than they would otherwise.

5 AN EXAMPLE PRODUCT-LINE

To illustrate the concepts of the previous sections, we present a GenVoca product-line model of graph traversal applications (Smaragdakis and Batory, 1998). This application domain is interesting because of the interchangeability of its components. More complex examples are discussed in Section 6 (Related Work).

The Model. A graph traversal application is a program that implements traversals on graphs. The library of refinements that we consider focuses on two traversals: vertex numbering and cycle checking. (The library can be expanded with other traversals—see (Smaragdakis and Batory, 1998)). The membership of our traversal library is:

```
undirected          // undirected graphs
directed            // directed graphs
dft<G x>            // depth-first traversal
bft<G x>            // breadth-first traversal
number<G x>         // vertex numbering
cycle<G x>          // cycle checking
```

where all members implement the (empty) interface `G`.

The mixin-layers `undirected` and `directed` implement undirected graphs and directed graphs, respectively. Both encapsulate a pair of classes `Vertex` and `Graph`. The methods of these classes support vertex addition and removal from graphs, but not traversals. Both mixin-layers are designed to implement the same interface, so that they are plug-compatible and interchangeable.

The mixin-layers `dft<G x>` and `bft<G x>` implement depth-first and breadth-first traversals, respectively. Both encapsulate refinements of the `Vertex` and `Graph` classes and add new abstract class `WorkSpace`. (Thus the `dft` and `bft` mixin-layers have three inner classes, two of which are mixins). The `Vertex` and `Graph` classes are refined with the addition of traversal methods: `GraphSearch` is added to `Graph` and `VertexSearch` is added to `Vertex`. Both methods take a `WorkSpace` object as a parameter. At various times during a graph or vertex search, e.g., prior to visiting a node and after visiting a node, a dispatch is made to the `WorkSpace` object for graph-traversal-specific actions. Each `WorkSpace` object supports three abstract methods: `init_vertex` (to initialize a vertex for a particular traversal), `preVisitAction`, and `postVisitAction`.

The mixin-layers **`number<G x>`** and **`cycle<G x>`** implement vertex numbering and cycle checking, respectively. Both encapsulate refinements of the **`Vertex`** and **`Graph`** classes, in addition to adding a subclass to **`WorkSpace`**. (**`number`** adds the subclass **`WorkSpaceNumber`**; **`cycle`** adds the subclass **`WorkSpaceCycle`**). **`number`** refines **`Graph`** by adding the **`VertexNumber`** method (which is called by application users to invoke vertex numbering); **`number`** refines **`Vertex`** by adding a public integer called **`vertexCount`** (which holds the assigned number of a vertex). The **`WorkSpaceNumber`** class supplies methods for initializing a vertex (i.e., setting **`vertexCount`** to zero), doing nothing for a **`preVisitAction`**, and assigning a number to a vertex for the **`postVisitAction`**. The **`cycle`** mixin-layer encapsulates similar capabilities and refinements for cycle-checking.

The Product-Line. Consider the following type equations:

```
frame1 = dft<undirected>
inst11 = number<frame1>
inst12 = cycle<frame1>
inst13 = number<cycle<frame1>>
```

A framework is defined by **`frame1`**: it is a set of classes that encapsulate a depth-first traversal on undirected graphs. These classes are incomplete in that there is no traversal application; an application must be supplied by extending these classes in a framework instance. Distinct framework instances are defined by **`inst11`**—**`inst13`**. A vertex numbering application is defined by **`inst11`**; a cycle checking application is defined by **`inst12`**; **`inst13`** defines an application that supports both vertex numbering and cycle checking. Note that one of the limitations of frameworks is exposed by this example: two different framework instances share common code (e.g., **`inst11`** and **`inst13`** share the **`number`** component; **`inst12`** and **`inst13`** share the **`cycle`** component). By encapsulating domain features as mixin-layers, we minimize code replication through component reuse.

Now consider the equations:

```
frame2 = dft<directed>
inst21 = number<frame2>
inst22 = cycle<frame2>
inst23 = number<cycle<frame2>>
```

A framework is defined by **`frame2`**: it is a set of classes that encapsulates a depth-first traversal on directed graphs. The framework instances **`inst21`**—**`inst23`** respectively define applications for vertex numbering, cycle checking, and both numbering and cycle checking on directed graphs. Note that the other limitation of frameworks is exposed by this example: the variation of classes in a framework (e.g., **`frame1`** and **`frame2`** are distinct frameworks that share the **`dft`** component). Other variations can be created by swapping **`dft`** with **`bft`**. This would yield vertex numbering and cycle checking applications on directed graphs using breadth-first search algorithms.[6]

Note that our frameworks above encapsulated a pair of features: a graph encoding and a traversal; a framework instance added the traversal application(s). We could have “raised the delineation line” so that our framework merely encoded a graph; instances of this framework would have to provide a traversal method and application:

```
frame0 = undirected;
inst01 = number<dft<frame0>>   // equivalent to inst11
inst02 = cycle<bft<frame0>>
```

The advantages of this decision is that the framework is more general, but writing instances is more work. Alternatively, we could have “lowered the delineation line” to enrich the capabilities of our framework:

```
frame00 = number<dft<undirected>> // same as inst11
inst01 = cycle<frame00>           // same as inst13
```

The advantages of this decision is that less code needs to be written in framework instances at an expense that the framework may be too specialized to use for a particular application. Mixin-layers, however, eliminates the annoying problem of deciding where to draw the framework-instance “line”; application designers are free to define the contents of frameworks as they see fit. This provides designers more flexibility in customizing their applications than using frameworks whose designs are inflexible to such customizations.

6 RELATED WORK

Use of Mixin-Layers. Product-lines using mixin-layer components have been created for extensible compilers (Batory, Lofaso, Smaragdakis, 1998; Findler and Flatt, 1998) and command-and-control simulators for fire support for the U.S. Army (Batory, Johnson, MacDonald, and von Heeder,

[6] Note that there is a definite order in which components can be legally composed: a directed/undirected graph component can be refined by a depth/breadth-first component, which can be refined by one or more traversal applications. Our typing of these components does not encode these constraints. See (Smaragdakis and Batory, 1998; Batory and Geraci, 1997) for constraint enforcement.

2000). Non-OO implementations of early versions of mixin-layers can be found in product-lines for databases, file systems, and network protocols (Batory and O'Malley, 1992; Heidemann and Popek, 1994).

Library Scalability. The drawbacks that we have noted with frameworks are classical examples of the *library scalability problem* (Batory, Singhal, Sirkin, and Thomas, 1993; Biggerstaff, 1994). The idea is simple: in a domain where there are *n* optional features, there can be in excess of *n!* different programs, each implementing a unique combination of features. Library components should *not* implement combinations of features, because (obviously) libraries would have exponentially large memberships. A better approach is to populate libraries with building blocks that implement *individual* features, and compose these blocks to synthesize the program with the desired combination of features. Such libraries are *scalable*: they grow at a linear rate, but the number of programs that can be synthesized from component combinations grows at an exponential rate.

We have seen that the classes of a framework may correspond to a composition of primitive refinements (i.e., mixin-layers). The classes of a framework instance may also correspond to a composition of primitive refinements. Since both are treated as encapsulated units, any variation made to either (corresponding to the addition, removal, or replacement of one or more refinements) will theoretically lead to an exponential number of variations to maintain. Our contribution in this paper is to show how to solve the library scalability problem for frameworks and framework instances.

Parameterized Components. GenVoca is an example of a programming paradigm called *parameterized programming*—that applications are synthesized by composing components via parameter instantiation (Goguen, 1986). It is interesting to note that programming support for mixin-layers is presently limited in Java. The most well-known versions of Java that offer parameterization (e.g., Pizza and GJ (Odersky and Wadler, 1997; Bracha, Odersky, Stoutamire, and Wadler, 1998)) do not support parameterized inheritance. This led to the development of JTS, a tool suite for creating a product-line of Java dialects, which does support parameterized inheritance (Batory, Lofaso, and Smaragdakis, 1998).

Most work on parameterized programming deals with parametric source code. Industry prefers to distribute binaries rather than source. Our approach presently cannot extend binary components, where the source code of mixin-frameworks is unavailable. This is a temporary problem. The static parameterizations of mixin-layers are simple enough to be expressed as parameterized binaries (e.g., parameterized Java .class files). That is, parameter instantiation is accomplished at class load time rather than at mixin-layer compile time. Recent work on parameterized Java class binaries suggests this possibility is not far off (Duncan and Hölzle, 1999). Thus, we anticipate in the future that libraries of binaries will be distributed.

Aspect-Oriented Programming (AOP). AOP is intimately related to refinements. An *aspect* is feature of a domain whose implementation "cross-cuts" multiple application classes (Kiczales, et. al., 1997). When an aspect is added to an existing application, multiple classes must updated. Clearly, aspects are refinements. An application-specific AOP implementation can provide custom refinements to any existing application. GenVoca implementations, on the other hand, start by conceptually decomposing legacy applications and resynthesizing them in an extensible way through component composition. Interface conformance plays a prominent role in the software composition process of GenVoca, more so than in AOP.

Reuse Contracts. The problems of framework version proliferation and architectural drift may be mitigated through formal annotations on classes (Codenie, De Hondt, Steyaert, and Vercammen, 1997). Reuse contracts record the design intentions of reusable classes and the assumptions made by actual users of those classes. Automated annotation checking can detect if new modifications violate the contract from either the producer or consumer point of view. Codifying the management of framework evolution in this way limits the proliferation of code that violates reuse conventions.

GenVoca is "neutral" on the use of contracts. Contracts can be used in the development and composition of mixin-layers. So the benefits accrued by using contracts are also available to layered designs.

Framework Coding Techniques. The use of traditional object-oriented construction techniques and patterns is typically not restricted when building frameworks under GenVoca. For example, inverting control through the use of hook methods (Fayad and Schmidt, 1997) is common in traditionally constructed frameworks and can easily encapsulated as GenVoca components. Many of the design choices that current framework developers face still need to be addressed when using GenVoca. For instance, the choice between black box and white box framework implementations must still be made when using a layered approach.

7 CONCLUSIONS

A framework is an object-oriented code-structuring technique that seems ideal for product-lines. The classes of a framework encapsulate the common algorithms that arise in a family of related applications. A particular application of a product-line is created by defining an instance of this framework, i.e., supplying concrete subclasses of framework classes to provide the necessary customizations. While frameworks are indeed useful, we and others have noticed that frameworks fail miserably in the very common case of optional features. Framework classes can vary (which leads to framework proliferation); classes of different framework instances can have much in common (which leads to code replication).

The core of these problems is that frameworks exhibit a rather inflexible design: the delineation between the content of framework classes and classes of framework instances is fixed. Application features are either hard-coded into framework classes or hard-coded into instance classes. In this paper, we have outlined a different approach for product-line implementation. We have presented a component-based model that reveals the building blocks of frameworks and framework instances. Our components allow application designers to define the set of features that they want both in their framework classes and instance classes. If features need to be changed, our model supports this by adding, swapping, or removing components from previously defined compositions. In general, the application product-lines that we can express with our model is much more varied with far less code replication than that which can be expressed by frameworks.

The essence of our approach is understanding software in terms of object-oriented collaborations or refinements, and creating a parametric model of product-lines that is based on refinement/collaboration composition. Many product-lines have been built using this approach before; the contribution of this paper is demonstrating that this approach offers significant advantages over frameworks in building application product-lines.

Acknowledgments. We thank Mike Kistler for his comments on earlier drafts of this paper.

8 REFERENCES

D. Batory and S. O’Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, *ACM TOSEM*, October 1992.

D. Batory, V. Singhal, M. Sirkin, and J. Thomas, “Scalable Software Libraries”, *ACM SIGSOFT 1993*.

D. Batory and B.J. Geraci, “Component Validation and Subjectivity in GenVoca Generators”, *IEEE Trans. Software Engineering*, February 1997.

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

D. Batory, “Product-Line Architectures”, Invited presentation, Smalltalk und Java in Industrie and Ausbildung, Erfurt, Germany, October 1998.

D. Batory, C. Johnson, R. MacDonald, and D. von Heeder, “Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study”, *International Conference on Software Reuse,* Vienna, Austria, June 2000.

T. Biggerstaff, “The Library Scaling Problem and the Limits of Concrete Component Reuse”, *International Conference on Software Reuse*, Rio de Janeiro, November 1-4, 1994, 102-110.

J. Bosch, “Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study”, *Software Architecture*, Kluwer Academic Publishers, 1999.

G. Bracha and W. Cook, “Mixin-Based Inheritance”, *ECOOP/OOPSLA 90*, 303-311.

G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler, “Making the future safe for the past: Adding Genericity to the Java Programming Language”, *OOPSLA 98*, Vancouver, October 1998.

W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen, “From Custom Applications to Domain-Specific Frameworks”, *Communications of the ACM*, 40(10), October 1997.

S. Cohen and L. Northrop, “Object-Oriented Technology and Domain Analysis”, *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

K. Czarnecki and U.W. Eisenecker, “Components and Generative Programming”, *SIGSOFT 1999*, LNCS 1687, Springer-Verlag, 1999.

J-M. DeBaud and K. Schmid, “A Systematic Approach to Derive the Scope of Software Product Lines”, *ICSE 99*.

A. Duncan and U. Hölzle, “Load-Time Adaptation: Efficient and Non-Intrusive Language Extension for Virtual Machines”, Technical Report TRCS99-09, University of California, Santa Barbara, 1999.

M. Fayad and D. Schmidt, “Object-Oriented Application Frameworks”, *Communications of the ACM*, 40(10), October 1997.

R.B. Findler and M. Flatt, “Modular Object-Oriented Programming with Units and Mixins”, *ICFP 98*.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

J.A. Goguen, “Reusing and Interconnecting Software Components”, *IEEE Computer*, February 1986.

H. Gomaa et. al., “A Prototype Domain Modeling Environment for Reusable Software Architectures”, *3rd International Conference on Software Reuse*, Rio de Janeiro, November 1-4, 1994, 74-83.

A.N. Habermann, L. Flon, and L. Cooprider, “Modularization and Hierarchy in a Family of Operating Systems”, *CACM*, May 1976.

M.G. Hayden, “The Ensemble System”, Ph.D. dissertation, Dept. Computer Science, Cornell, January 1998.

J.S. Heidemann and G.J. Popek, “File-System Development with Stackable Layers”, *ACM Transactions on Computer Systems*, 12(1), 58-89, 1994.

R. Johnson and B. Foote, “Designing Reusable Classes”, *Journal of Object-Oriented Programming,* 1(2): June/July 1988, 22-35.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, *ECOOP 97*, 220-242.

D. McIlroy, “Mass Produced Software Components”, *Software Engineering: Report on a Conference by the Nato Science Committee*, October 1968, P. Naur and B. Randell, eds. 138-150.

M. Odersky and P. Wadler, “Pizza into Java: Translating Theory into Practice”, *ACM Symposium on Principles of Programming Languages 1997*, 146-159.

D.L. Parnas, “On the Design and Development of Program Families”, *IEEE Transactions on Software Engineering,* March 1976.

H. Partsch and R. Steinbruggen, “Program Transformation Systems”, *Computing Surveys*, March 1983.

T. Reenskaug, et al., “OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems”, *Journal of Object-Oriented Programming*, 5(6): October 1992, 27-41.

Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”, *ECOOP 1998*.

M. VanHilst and D. Notkin, “Using C++ Templates to Implement Role-Based Designs”, *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.

D.M. Weiss and C.T.R. Lai, *Software Product-Line Engineering*, Addison-Wesley, 1999.

Generating Product-Lines of Product-Families

Don Batory, Roberto E. Lopez-Herrejon, Jean-Philippe Martin
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
{batory, rlopez, jpmartin}@cs.utexas.edu

Abstract

GenVoca is a methodology and technology for generating product-lines, i.e. building variants of a program. The primitive components from which applications are constructed are ***refinements*** *or* ***layers****, which are modules that implement a feature that many programs of a product-line can share. Unlike conventional components (e.g., COM, CORBA, EJB), a layer encapsulates fragments of multiple classes. Sets of fully-formed classes can be produced by composing layers. Layers are modular, albeit unconventional, building blocks of programs.*

But what are the building blocks of layers? We argue that facets is an answer. A facet encapsulates fragments of multiple layers, and compositions of facets yields sets of fully-formed layers. Facets arise when refinements scale from producing variants of individual programs to producing variants of multiple integrated programs, as typified by ***product families*** *(e.g., MS Office).*

We present a mathematical model that explains relationships between layers and facets. We use the model to develop a generator for tools (i.e., product-family) that are used in language-extensible ***Integrated Development Environments (IDEs)****.*

1. Introduction

Over the last thirty years, program modularity has been dominated by *object-orientation (OO)*: method, class, and package encapsulations are standard concepts. Over this same period, another form of program modularity has arisen. The concept is *feature refinement* — that is, a module that encapsulates the implementation of a *feature*, which is a product characteristic that customers view as important in describing and distinguishing programs within a family of related programs (e.g., a product-line) [15].

Feature refinement is a very general concept and many different implementations of it have been proposed, each with different names, capabilities, and limitations: layers [2], features [18], collaborations [25, 35, 21], subjects [16], aspects [19], and concerns [33]. Unlike traditional component technologies (such as COM, CORBA, and EJB), a feature refinement encapsulates not an entire method or class, but rather fragments of methods and classes. Figure 1 depicts a package of three classes, **`c1`**—**`c3`**. Refinement **`r1`** *cross-cuts* these classes, i.e., it encapsulates fragments of **`c1`**—**`c3`**. The same holds for refinements **`r2`** and **`r3`**. Composing refinements **`r1`**—**`r3`** yields a package of fully-formed classes **`c1`**—**`c3`**. Because refinements reify levels of abstraction, feature refinements are often called *layers* — a name that is visually reinforced by their vertical stratification of **`c1`**—**`c3`** in Figure 1. As refinements, layers, and features are so closely related, their terms are often used interchangably. In general, layers are modular, albeit unconventional, building blocks of programs.

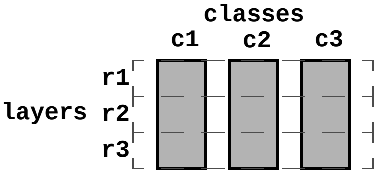


Figure 1. Classes and Refinements (Layers)

This raises an interesting question: if layers (features) are the building blocks of programs, what are the building blocks of layers (features)? We argue that an answer is a *facet*. The idea is simple: Figure 2 depicts a set of three layers, **`r1`**—**`r3`**. Facet **`f1`** *cross-cuts* these layers, i.e., it encapsulates fragments of **`r1`**—**`r3`**. The same for facets **`f2`** and **`f3`**. Composing facets **`f1`**—**`f3`** yields fully-formed layers **`r1`**—**`r3`**. Although it appears that Figure 2 is just Figure 1 turned on its side, where classes and facets are indistinguishable, this is not the case. Facets are *not* classes.

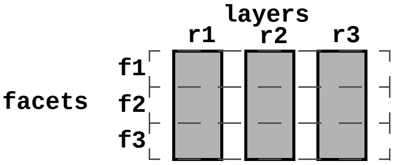


Figure 2. Facets

A year ago, we would not have believed facets to exist or, if they did, to have any utility. To our surprise, we now believe that they are very common. Facets arise when feature refinements scale beyond the confines of an individual program or package. As a perspective, contemporary models of feature refinements allow clients to customize *individual* programs; the set of all program variants that can be produced is a *product-line*. In contrast, a *product-family* is an integrated *suite* of programs, each program having different capabilities [9]. Microsoft Office is an example; it includes the Excel (spreadsheet), Word (text processor), and Access (database) programs. Given how common product-families are, an interesting question is: can feature refinements scale to define a product-line of product-families?

In this paper, we present new results on feature refinement modularity. We show that refinements do scale to product-families and there are interesting twists in doing so. Previously considered "atomic" refinements are revealed to be composed of more elementary refinements called *gluons*. Gluons are arranged in regular ways to form both "atomic" refinements and facets. We present a model of gluons, called *Origami*, that reveals software to have an elegant mathematical structure that leads to simpler designs and more powerful models of code generation.

We base our work on GenVoca, a methodology and technology for generating product-lines using feature refinements. This paper shows how GenVoca ideas scale to product-families, something that has not been demonstrated previously. Further, we argue that our results are directly applicable to other models, such as *Aspect-Oriented Programming (AOP)* [20] and *Multi-Dimensional Separation of Concerns (MDSC)* [33, 23, 24], and thus are not GenVoca-specific. We explore this connection further in Related Work. We begin with a motivating example that illustrates the phenomenon of facets.

2. A Motivating Problem

An *Integrated Development Environment (IDE)* is a suite of applications (i.e., a product-family) that allow users to write, debug, visualize, and document programs. Among the programs, here called *tools*, of an IDE are a compiler, debugger, editor, formatter, and document generator (e.g, Javadoc). Figure 3a depicts some of these tools, each of which is implemented in a different package.

The problem we consider is generating IDE tools that all work on the same language dialect or *Domain-Specific Language (DSL)*. The use of DSLs have shown benefits in terms of understandability, maintainability, and extensibility in software design and development processes [11]. Providing IDE tools to support DSL program compilation, editing, debugging, and document generation is essential for the successful adoption of DSL technology. In particular, our work focuses on dialects of Java.

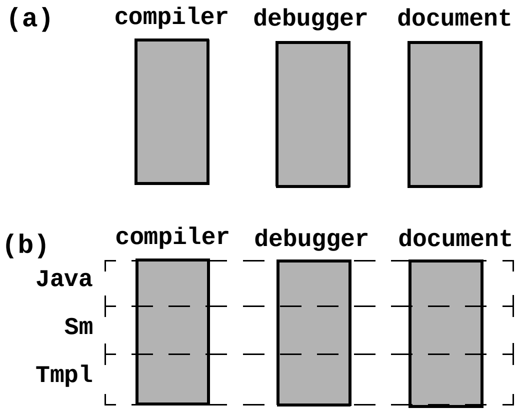
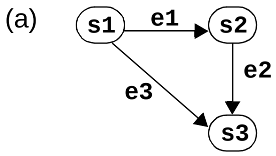


Figure 3. IDE Tools and Cross-Cutting Language Features

An example of a Java dialect is the one we are using to write fire-support simulators for the U.S. Army [6]. As a brief summary, fire support programs are a set of collaborating state machines. Figure 4a depicts a state machine of three states (**s1**, **s2**, **s3**) and three edges (**e1**, **e2**, **e3**), where an edge denotes a transition from one state to another. For example, edge **e3** begins at state **s1** and ends at state **s3**. Wide spectrum languages, like Java, are typically used to implement state machines. The resulting code, even when using the state machine design pattern [14], is often ugly, involving nested switch statements, large numbers of methods or classes. This places a burden on maintenance engineers because they must re-engineer the simple abstractions of state machines (e.g., Figure 4a) from the code in order to understand and modify it. In contrast, Figure 4b shows the specification of Figure 4a in our extended Java language. Highlighted are state declarations and edge declarations. We have found that state-machine-extended Java programs are about half the size of their pure-Java counterparts, and this in turn simplifies the writing, maintenance, and understanding of domain-specific programs. Similar benefits accrue when other extensions, such as templates, are added to Java.

In the future, we expect to work in other domains, each requiring their own specific extensions to Java. This means that we need to be able to construct IDE tools targeted for a particular Java dialect, or more generally, we need to define a product-line for a product-family of IDE tools. The novelty of our work is that we are using refinements (layers) as the unit of modularity.

Figure 3b revisits our IDE tools, but this time we expose the layers from which they were constructed. One layer, **Java**, encapsulates a cross-cut of the compiler, debugger, and document generation packages that is specific to the Java language. A second layer, **Sm**, encapsulates another cross-cut of these tools; the encapsulated code fragments



```
(b) state_machine example {
       event_delivery receive_message(M m);
       no_transition { error( -1, m ); }
       otherwise_default { ignore_msg(m); }

       states s1, s2, s3;                    states

       edge e1 : s1 -> s2                    edges
       conditions !booltest() do
          { /* e1 action */ }

       edge e2 : s2 -> s3
       conditions booltest() do
          { /* e2 action */ }

       edge e3 : s1 -> s3
       conditions true  do
          { /* e3 action */ }

       // Java class data members and
       // methods from here

       boolean booltest() { ... }
       example() { current_state = start; }
    }
```

Figure 4. State Machines in Extended Java

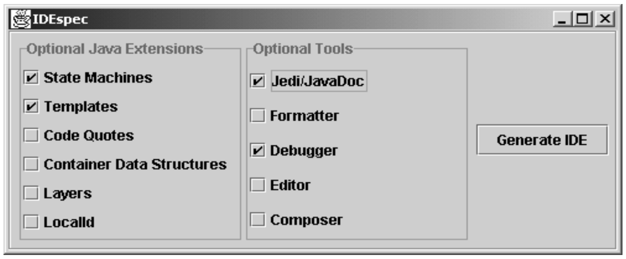
implement our state machine extension to Java. (That is, `Sm` extends the compiler tool to compile state machine specifications, it also extends the debugger so that it can debug state machine programs, etc.) A third layer, `Tmpl`, encapsulates the code fragments that implement our template extension to Java.

In principle, this is encouraging: layers (features) scale to product-families. That is, refinements scale to the encapsulation of fragments of multiple tools. Further, it appears that an IDE tool generator has a simple, declarative GUI front-end. Figure 5 suggests its basic outline: a client selects a set of optional language features and a set of tools (as not all might be needed), and by pressing the `Generate` button, the generator produces the requested set of IDE tools to work on the specified dialect of Java.

While the GUI is simple, the technology that underlies this generator is sophisticated. To understand how it works, we first review the GenVoca model and the Jakarta Tool Suite.

3. GenVoca

GenVoca is a design methodology for creating product-lines and building architecturally-extensible software — i.e., software that is extensible via component additions and removals. GenVoca is an outgrowth of an old and practitioner-ignored methodology called *step-wise refinement* [12], which asserts that efficient programs can be created by revealing implementation details in a progressive manner. Traditional work on step-wise refinement has focussed on microscopic program refinements (e.g., `x+0` ⇒ `x`), for which one had to apply hundreds or thousands of refinements to yield admittedly small programs. While the approach is fundamental and industrial infrastructures are on the horizon [7, 27], GenVoca extends step-wise refinement by scaling refinements to a multi-class-crosscut granularity, so that each refinement adds a feature to a program, and composing a few refinements yields an entire application.



Optional Java Extensions

☑ State Machines
☑ Templates
☐ Code Quotes
☐ Container Data Structures
☐ Layers
☐ LocalId

Optional Tools

☑ Jedi/JavaDoc
☐ Formatter
☑ Debugger
☐ Editor
☐ Composer

Figure 5. A GUI for an IDE-Tool Generator

3.1. Model Concepts

The central idea is programs are *values* and that refinements are *functions* that add features to programs. Consider the following constants that represent programs with different features:

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

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

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

A multi-featured application is specified by an *equation*, i.e., a named composition of functions. Different equations define a family of applications, such as:

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

Thus, by casually inspecting an equation, one can determine features of an application.

Note that there is a subtle but important confluence of ideas: a function represents both a feature *and* its implementation — there can be different functions that offer different implementations of the *same* feature:

```
k1(x)      // adds feature k with
           // implementation1 to x
k2(x)      // adds feature k with
           // implementation2 to x
```

When an application requires feature **k**, it is a problem of *equation optimization* to determine which implementation of **k** would be the best (e.g., provide the best performance)[1]. It is possible to automatically design software (i.e., produce an equation that optimizes some quantitative criteria) given a set of declarative constraints for a target application. An example of this kind of automated reasoning is in [5].

Although GenVoca constants and functions appear to be untyped, typing constraints do exist in the form of design rules. *Design rules* capture syntactic and semantic constraints that govern the legal composition of features [3]. It is not unusual that the selection of a feature will disable (or enable) the selection of other features. For this paper, design rules constrain the order in which features are composed. Details of their specification are beyond the scope of this paper and can be found in [3].

3.2. Model Implementation

Feature refinements are intimately related to *collaboration-based designs* [25, 30, 28]. A *collaboration* is a generic relationship among multiple classes. An individual class represents a particular role in a collaboration, and is a set of data members, methods, and method overrides that are needed to carry out this role. Because collaborations are defined largely in isolation of each other, they define features that are reusable, i.e., that can be used in the construction of many applications. A particular application is a composition of collaborations. Each class of an application plays one or more roles, where each role originates from a different collaboration.

A GenVoca constant is a set of classes. Figure 6 depicts a constant **i** with four classes ($\mathbf{a}_i$—$\mathbf{d}_i$). A GenVoca function is a set of classes and class refinements. A *class extension* is a subclass[2]: it encapsulates new data members, methods, and method overrides of its parent class. Figure 6 shows the result of applying function **j** to **i**: three classes are extended and another class is added. (That is, **j** encapsulates a cross-cut that includes classes $\mathbf{a}_j$, $\mathbf{c}_j$, and $\mathbf{d}_j$ that extend $\mathbf{a}_\mathtt{i}$, $\mathbf{c}_\mathtt{i}$ and $\mathbf{d}_\mathtt{i}$ respectively, and adds class $\mathbf{e}_j$). Figure 6 also shows the application of function **k** to **j(i)**, resulting in two classes being extended. In general, a forest of inheritance hierarchies is created as layers are composed, and this forest grows progressively broader and deeper as the number of layers increase [4].

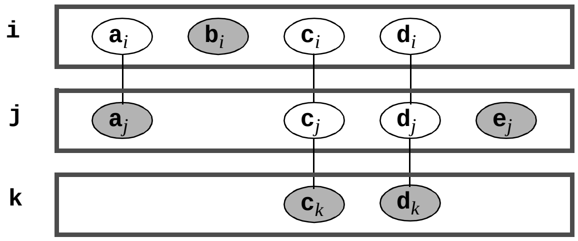


Figure 6. Implementing Refinements as Collaborations

Linear inheritance chains, called *refinement chains*, are common in this implementation method. The general rule is that only the bottom-most class of a refinement chain is instantiated, because this class implements all roles that were assigned to it. These classes are shaded in Figure 6.

For example, the bottom-most class of the **c** refinement chain plays the roles $\mathbf{c}_i$, $\mathbf{c}_j$, and $\mathbf{c}_k$ in the collaborations **i**, **j**, and **k** respectively.

Because GenVoca functions may be composed in arbitrary orders, class extensions can be implemented as mixins (modulo footnote 2). A *mixin* is a template: it is a class whose superclass is specified via a parameter. Mixins enable the order in which subclasses appear in a refinement chain to be permuted. More details on mixins and implementing collaborations as mixins are discussed elsewhere [30, 28, 13].

4. The Jakarta Tool Suite (JTS)

The *Jakarta Tool Suite (JTS)* is a suite of compiler-compiler tools for building families of language translators [4] that we used to implement our IDE tools. A language family is defined by a GenVoca model, which consists of a single constant — the base language — and functions that define optional extensions to the base. The family of Java dialects that can be synthesized by JTS is a GenVoca model, named **J,** consisting of the **Java** constant (representing the Java 1.4 language) and functions that add to Java embedded domain-specific languages for state machines (**Sm(x)**), container data structures (**P3(x)**), code fragments a la Lisp quote and unquote (**Ast(x)**), hygienic macros (**Gscope(x)**), and templates (**Tmpl(x)**), among others [4, 5, 29]:

```
J = { Java, Sm(x), P3(x), Ast(x), Gscope(x),
      Tmpl(x), ...}                           (1)
```

A particular dialect of Java is defined by an equation. The current dialect is called **Jak** (short for Jakarta), which is Java extended with state machines and templates:

[1] Different equations represent different programs and equation optimization is over the space of semantically equivalent programs. This is identical to relational query optimization: a query is initially represented by a relational algebra expression, and this expression is optimized. Each expression represents a different, but semantically equivalent, query-evaluation program as the original expression.

[2] More accurately, a class extension is a subclass that assumes the name of its parent class. This is different than typical subclassing, but is a simple way in which the contents of a class can be refined.

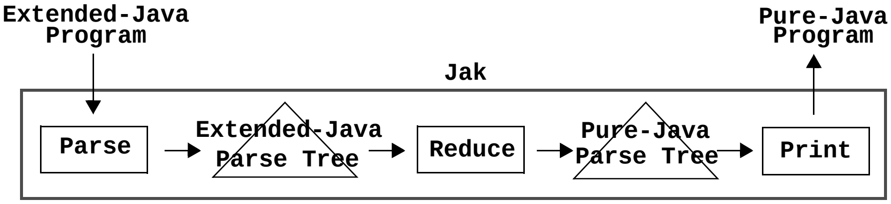


Figure 7. The Organization of the Jak Translator

```
Jak = Sm( Tmpl( Java ))                        (2)
```

The state machine and template features are independent of each other. As a consequence, the order in which **Sm** and **Tmpl** are composed doesn't matter. Thus, an equation equivalent to **(2)** is:

```
Jak = Tmpl( Sm( Java ))                        (3)
```

JTS converts such equations directly into a Java package using the ideas of Section 3 to implement a translator (preprocessor) for that language. The **Jak** preprocessor, like other JTS-produced preprocessors, translates an extended-Java program (with state machines and templates) into a program that represents its pure-Java counterpart. Figure 7 depicts its organization. An extended-Java program is parsed into an extended-Java parse tree. A reducer walks the tree, replacing each non-Java node or subtree with its pure-Java counterpart. The result is a pure-Java parse tree, which is then printed. The printed program is the Java translation of the extended-Java program. No matter what language extensions are added to Java, the organization of Figure 7 remains the same. This organization was inspired by Microsoft's IP [27].

Jak is only one of a number of IDE tools that must be customized to a particular language dialect. Another is a Javadoc-like tool that harvests comments from specific program constructs and displays them neatly on HTML pages. Obviously, Sun Microsystem's Javadoc [17] can't be used directly, as it only understands pure-Java programs (and documenting generated pure-Java programs typically isn't all that useful). So we created a language extensible version of Javadoc called *Jedi (Java Extensible DocumentatIon)*. **Jedi**, like **Jak**, is produced by JTS using a GenVoca model, called **D**. The lone constant is **JavaDoc**, which encapsulates the code that parsers, harvests, and documents comments in pure-Java programs. Functions of this model extend **JavaDoc** with the capabilities of producing HTML documentation for state machines (**SmDoc(x)**), templates (**TmplDoc(x)**), etc. In principle, the elements of models **J** and **D** are in one-to-one correspondence: each language extension in **J** has a corresponding documentation extension in **D**.[3]

```
D = { JavaDoc, SmDoc(x), TmplDoc(x), ... } (4)
```

A particular version of **Jedi** is specified as an equation:

```
Jedi = SmDoc( TmplDoc( JavaDoc ))          (5)
```

As before, the template and state machine layers of **Jedi** are independent, and thus can be composed in any order. Thus, an equation equivalent to **(5)** is:

```
Jedi = TmplDoc( SmDoc( JavaDoc ))          (6)
```

Figure 8 depicts the internal organization of **Jedi**. An extended-Java program is parsed into an extended-Java parse tree. A harvester walks the tree, harvesting comments prefacing particular language constructs (e.g., class, method, state machine declarations, etc.) and stores them in a comment repository. Finally, a doclet reads the contents of the comment repository, and formats harvested comments neatly on an HTML page, which users recognize as Javadoc-like output.

It is interesting to note that **Jak**, **Jedi**, and other IDE tools can be expressed directly by a single GenVoca model, **IDE_Model**, where different equations correspond to different tools. The primitives of this model are *tool features*. There is a lone constant **Parse**, which represents the parser for the given language dialect, and there are functions for reducing extended-Java constructs to pure-Java (**Reduce(x)**), for printing parse trees (**Print(x)**), for harvesting comments from parse trees (**Harvest(x)**), for

[3] In practice, **J** and **D** need not be in correspondence. That is, there might be a language extension without a corresponding documentation extension, simply because that extension has yet to be built.

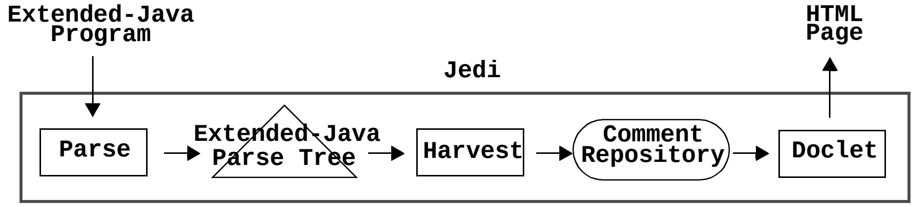


Figure 8. The Organization of the Jedi Translator

producing HTML documents from harvested comments (**Doclet(x)**), and so on.

```
IDE_Model = { Parse, Reduce(x), Print(x),
         Harvest(x), Doclet(x), ... }      (7)
```

Each IDE tool has an equation. The equations for **Jak** and **Jedi** are given below:

```
Jak  = Print( Reduce( Parse ))              (8)
Jedi = Doclet( Harvest( Parse ))            (9)
```

Even though the above equations look suspiciously like “functional” (e.g. Haskell) programs, they really do represent a composition of features that are implemented by the usual cross-cuts. Figure 9a shows the **Parse** layer to encapsulate a set of parser classes (only one class is shown), a set of parse tree node classes (again, only one is shown), and a **Main** class. The **Reduce** layer extends each parse tree node type with a reduction method (specific to that type), and extends the **Main** class with a call to reduce an extended-Java parse tree to a pure-Java parse tree. Finally, the **Print** layer extends each parse tree node type with a print method (specific to that node type) and extends the **Main** class with a call to print the reduced tree. Again, the terminals of the resulting refinement chains are the classes that are instantiated. Figure 9b shows the code added by each layer to the **Main** class.

> Note: the order in which tool features are composed is important. **Parse** must be first, followed by **Reduce** and then **Print**, or followed by **Harvest** and then **Doclet**. The reason is **Harvest** extends classes in **Parse**, and **Doclet** references methods in **Harvest**. The same applies to **Reduce** and **Print**. These constraints are examples of design rules.

Language extensibility is *not* part of the **IDE_Model**. In fact, astute readers may have noticed that our original descriptions of **Jak** and **Jedi** were based on GenVoca models of *language features*, and not *tool features*. Clearly these models are related, but how? Further, we know the **Jak** equations **(2)** and **(8)** must be equivalent, and so too the **Jedi** equations **(5)** and **(9)**. But how? An answer requires a closer look at the internals of these tools, which we do in the next section.

5. Gluons

Language features are orthogonal to tool features. This means that we can understand the modularity of **Jak** and **Jedi** in terms of matrices, where rows correspond to language features and columns correspond to tool features.

The matrix for **Jedi** is shown in Figure 10. Each matrix entry lists the name of a module that implements *a particular tool feature for a particular language feature*. For example, **Sharvest** is a module that implements the harvesting of comments from state machine specifications. **Jharvest** harvests comments from Java specifications. **Tdoclet** formats comments from template declarations on an HTML page. And so on. A composition of these modules implements **Jedi**.

	Doclet	Harvest	Parse
Java	Jdoclet	Jharvest	Jparse
Sm	Sdoclet	Sharvest	Sparse
Tmpl	Tdoclet	Tharvest	Tparse

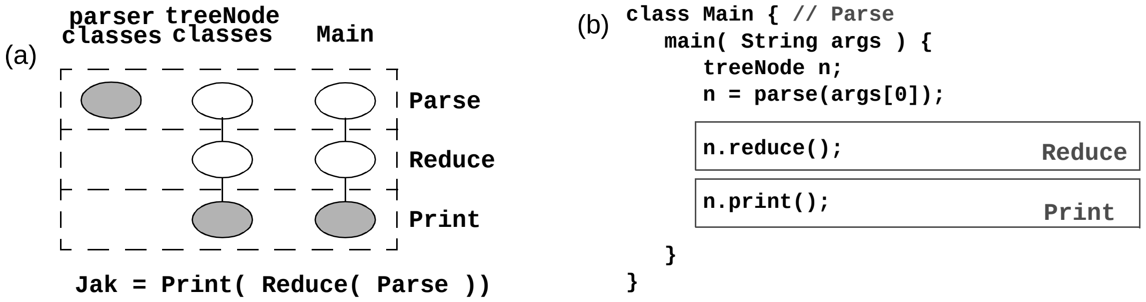
Figure 10. Jedi Matrix

The matrix for Jak is shown in Figure 11, and has a similar interpretation. There is a difference: there are no **Sm** and **Tmpl** row entries for the **Print** column. The reason is simple: consider the interpretation of **Sreduce**: it is a module that transforms parse trees on state machines into parse trees of pure Java. The **Jprint** module prints parse trees of pure Java. So once the **Sreduce** module performs its task, the **Jprint** module is invoked. Thus there is no need for a module that prints state machine parse trees. The same argument applies for templates. Once again, a composition of these modules implements the **Jak** tool.

	Print	Reduce	Parse
Java	Jprint	Jreduce	Jparse
Sm	–	Sreduce	Sparse
Tmpl	–	Treduce	Tparse

Figure 11. Jak Matrix

These matrices provide the first indication of facets. Let us call matrix entries *gluons* and consider the **Jedi** matrix of



```
class Main { // Parse
   main( String args ) {
      treeNode n;
      n = parse(args[0]);

      n.reduce();

      n.print();

   }
}
```

Figure 9. Cross-Cuts of Tool Features

Figure 10. Each row represents a language feature; its implementation is a composition of the gluons in that row. The **Java** language feature, for example, is defined by a composition of the **Jparse**, **Jharvest**, and **Jdoclet** gluons. The same for other rows.

By the same reasoning, each tool feature is represented by a column and is implemented by a composition of gluons in that column. For example, the **Harvest** tool feature is a composition of the **Jharvest**, **Sharvest**, and **Tharvest** gluons. The same for other columns.

Thus, if layers are rows of gluons, then facets are columns of gluons — columns cross-cut every row. Similarly, if layers are columns of gluons, then facets are rows of gluons — rows cross-cut every column. Thus, a facet is simply a feature along a dimension, and the implementation of a facet cross-cuts features of other dimensions.

Two questions remain. First, what are gluons? Very simply, they are elementary layers (refinements) that implement the intersection of pair of orthogonal features. Or more accurately, a gluon implements a *feature of a feature* or a *building block* of both a language feature and tool feature. A gluon is a module that encapsulates any number of classes and class extensions, and has straightforward implementation as a set of classes and mixins. Thus, we can represent each gluon as a GenVoca constant or function.

When we create the matrices of Figure 10 and Figure 11, we are decomposing a composite language feature (layer) or tool feature (layer) into more primitive layers — in essence, separating their concerns. The theoretical justification is simple: any function **F** can be the result of composing more primitive functions $F_1 \ldots F_n$, and any constant **C** could be the result of composing a more primitive constant **C′** with one or more functions $F_1' \ldots F_n'$:

$$
\begin{aligned}
F(x) &= F_1(\ F_2(\ \ldots\ F_n(x)\ \ldots\ ))\\
C &= F_1'(\ F_2'(\ \ldots\ F_n'(\ C'\ )\ldots))
\end{aligned}
$$

Decomposing software is modeled by decomposing equations.

Second, we want to represent **Jak** and **Jedi** as equations that are compositions of gluons. Equations for **Jak** and **Jedi** are:

```
Jak = Jprint( Treduce( Sreduce( Jreduce(
     Tparse( Sparse( Jparse ))))))        (10)
Jedi = Tdoclet( Tharvest( Tparse(
     Sdoclet( Jdoclet( Sharvest(
     Jharvest( Sparse( Jparse )))))))))   (11)
```

These equations are *much* more complex than those of previous sections. Two questions immediately arise: (a) are they *correct* — are they legal compositions of gluons? and (b) are they *consistent* — do they represent tools that work on the same language dialect? Existing design rule checking algorithms can validate these equations [3], but there are no algorithms to check for *consistency.* In fact, without the techniques presented in the next section, it would take some time to write such equations manually and verify that they are consistent. We would expect the consistency problem to be much more difficult for larger sets of tools and more complex language dialects. Hence, automated support is required to write these equations and to ensure their consistency: we need a model of gluons.

6. Origami: A Model of Gluons

The notation that we have used prior to this section is consistent with previous work on GenVoca. However, the usual “functional” notation becomes cumbersome as equations become complicated. So we make a cosmetic switch in notation to simplify our upcoming discussions. Without loss of generality, instead of writing **A = B(C(D))** we write **A = B o C o D**, where **o** is the (function) composition operator.

GenVoca models are inherently one-dimensional; they are sets of constants and functions. In contrast, models of gluons are 2-dimensional — and generally n-dimensional — and need to be treated accordingly.[4] Consider the matrix of Figure 12, called an *Origami matrix*, where rows denote language features and columns are tool features. Elements of this matrix are gluons.

Adding new entries to this matrix is easy. When a new row is added, a gluon must be supplied for every existing column. For example, to add the container data structure (**Ds**)

[4] All examples of gluons that are known to us can be expressed in 2-dimensions. We would expect examples yet to be discovered to have higher dimensionality. A 3-dimensional example would expose “facets of facets”, and so on. Thus our model scales.

	Doclet	Harvest	Parse	Reduce	Print	...
Java	Jdoclet	Jharvest	Jparse	Jreduce	Jprint	...
Sm	Sdoclet	Sharvest	Sparse	Sreduce	-	...
Tmpl	Tdoclet	Tharvest	Tparse	Treduce	-	...
Ds	Ddoclet	Dharvest	Dparse	Dreduce	-	...
...	...	...	...	...	...	...

Figure 12. An Origami Matrix

language feature, we would have to add `Dparse` (a parser for container DSL specifications), `Dreduce` (reduction methods to transform container specification parse trees to Java parse trees), `Dharvest` (a harvester of comments on container specifications), and `Ddoclet` (a doclet that formats container comments). Some entries (such as the entry for the `Print` column) are "empty" because no code needs to be written to implement that functionality. In such cases, the identity function (denoted by "`-`") is supplied.

Symmetrically, when a new column is added, a gluon must be supplied for every existing row. To add a new doclet that produces, say Word documents, we would add `Jword` (a doclet that formats Java comments in Word), `Sword` (a doclet that formats state machine comments in Word), `Tword` (a doclet that formats template comments in Word), and so on. Again, if no code needs to be written for a particular entry, an identity function is supplied.

An application (expression) is created by *folding* an Origami matrix (hence its name). Rows are folded together by composing the corresponding gluons in each column. Columns are folded together by composing the corresponding gluons in each row. Folding continues until a 1×1 matrix is produced; the entry of this matrix is the desired expression. (Unlike true Origami, rows and columns to be folded need not be adjacent. For our examples, we have arranged this matrix so that they are).

Rows and columns cannot be chosen at random for folding. Rows (columns) must be composed in design rule order. That is, if we are folding tool features, we must begin with the `Parse` column, and then fold/compose the `Harvest` column, and finally the `Doclet` column, just as design rules prescribe for the `IDE_Model`. Similarly, if we are folding language features, we must begin with the `Java` row, and then fold the `Sm` row and `Tmpl` rows in any order, as prescribed by the language feature models `J` and `D`. The reason for this is that language features and tool features are orthogonal.[5]

As an illustration, suppose we want to create an equation for current version of `Jedi`. We project this matrix of unnecessary rows and columns, leaving the rows for `Java`, `Sm`, and `Tmpl`, and the columns `Parse`, `Harvest`, and `Doclet` yielding Figure 13a. (Note that there can be different kinds of doclets — HTML, Word, etc. So part of this projection is selecting the appropriate tool features).

Figure 13b shows the result of composing the `Java` row with the `Sm` row. Figure 13c-d shows the result of composing the `Harvest` column with the `Parse` column, and this result with the `Doclet` column. A matrix of two rows and one column results. The final fold merges the remaining two rows to yield the expression of equation `(11)`. We leave it as an exercise for readers to discover the folding of equation `(10)`.

Other constraints may preclude certain foldings, but this is the essential idea. In the next section, we show how we can use Origami to produce sets of language-dialect consistent equations.

7. An Application of Origami

Recall the GUI for the IDE generator of Figure 5: users select a set of optional language features and a set of tools, and the generator produces this set of tools for the speci-

[5] In effect, gluons make our model more powerful without additional complexity as the existing row and column rules can be used without modification.

(a)	Doclet	Harvest	Parse
Java	Jdoclet	Jharvest	Jparse
Sm	Sdoclet	Sharvest	Sparse
Tmpl	Tdoclet	Tharvest	Tparse

(b)	Doclet	Harvest	Parse
Sm o Java	Sdoclet o Jdoclet	Sharvest o Jharvest	Sparse o Jparse
Tmpl	Tdoclet	Tharvest	Tparse

(c)	Doclet	Harvest o Parse
Sm o Java	Sdoclet o Jdoclet	Sharvest o Jharvest o Sparse o Jparse
Tmpl	Tdoclet	Tharvest o Tparse

(d)	Doclet o Harvest o Parse
Sm o Java	Sdoclet o Jdoclet o Sharvest o Jharvest o Sparse o Jparse
Tmpl	Tdoclet o Tharvest o Tparse

(e)	Doclet o Harvest o Parse
Tmpl o Sm o Java	Tdoclet o Tharvest o Tparse o Sdoclet o Jdoclet o Sharvest o Jharvest o Sparse o Jparse

Figure 13. Folding an Origami Matrix

fied language dialect.[6] To see how the generator works, we begin with the Origami matrix of Figure 12 and eliminated all language feature rows that were not selected. Figure 14 shows this matrix for the current **`Jak`**/**`Jedi`** dialect.

Rows are folded in design rule order (i.e., **`Tmpl o Sm o Java`**). In general, our generator simply uses design rules to hard-code this ordering. The result is a $1 \times n$ matrix (i.e. a row) in Figure 15. Note the row's semantics. First, each column defines an equation for a tool feature:

```
Doclet  = Tdoclet o Sdoclet o Jdoclet
Harvest = Tharvest o Sharvest o Jharvest
Parse   = Tparse o Sparse o Jparse
...
```

That is, the **`Doclet`** equation is a composition of gluons that builds a doclet layer for the Java language that has been extended by state machines and templates. The **`Harvest`** equation defines a harvest layer for the Java language that has been extended by state machines and templates, and so on. Thus, by folding rows in design-rule order, *we have produced a set of equations for tool features that are consistent with respect to a particular language dialect.*

Second, the row itself is exactly the set of tool features that comprise the **`IDE_Model`**. Since we know the **`IDE_Model`** equations for each tool (e.g., **`(8)`**,**`(9)`**), we use these equations and plug in the generated definitions for their tool features. Thus, for each GUI-selected tool, we evaluate its equation, and send it to a generator to produce the Java package for that tool. In this way, our IDE generator produces language-dialect-consistent tools from a simple declarative specification.

8. Implementation

We currently have five tools that are language-dialect sensitive: **`Jedi`**, **`Jak`**, **`Mixin`** and **`Jampack`** (two different tools that compose **`Jak`** specifications), and **`UnMixin`** (a tool that propagates changes made manually in composed code back to their defining "gluon" layers). There are nine language features and nine tool features that span these tools. (We support several template features as well as additional language features, such as hygienic macros). Table 1 lists for each tool its size in Java LOC, and the number of gluons that define its equation. Note that equations provide *very* compact specifications of these tools.

Just with this set of tools (for which we expect many more), we are generating over 100K LOC. Without Origami, our tool equations are seemingly randomly-ordered compositions of gluons that are difficult to understand and update. Origami imposes a regularity in equation organization and layer modularization that enables us to generate product-families from simple specifications. Just as important, it helps us control the complexity of feature-refinement-based representations of product-families.

Tool	# of Gluons	Size in Java LOC
Jak	14	26K
Jedi	31	32K
Mixin	23	24K
JamPack	22	26K
UnMixin	20	23K

Table 1. Size of Generated IDE Tools

9. Relevance to Other Technologies

There are many non-GenVoca examples of Origami. One possibility is the internationalization of programs made during Windows OS installations. By selecting a particular language (or dialect), the GUIs of different Windows programs are modified to present commands in that language. Origami also has relationships to component-based software design.

Microsoft's *Component Object Model (COM)*, Sun's *Enterprise Java Beans (EJB)*, and CORBA are conventional component models [26] that deal with *interface-based programming* — clients program to standardized

	Doclet	Harvest	Parse	Reduce	Print	...
Java	Jdoclet	Jharvest	Jparse	Jreduce	Jprint	...
Sm	Sdoclet	Sharvest	Sparse	Sreduce	-	...
Tmpl	Tdoclet	Tharvest	Tparse	Treduce	-	...

Figure 14. A Row-Projected Matrix

	Doclet	Harvest	Parse	Reduce	Print	...
Tmpl o Sm o Java	Tdoclet o Sdoclet o Jdoclet	Tharvest o Sharvest o Jharvest	Tparse o Sparse o Jparse	Treduce o Sreduce o Jreduce	Jprint	...

Figure 15. A Row-Folded Matrix

[6] We assume the set of language features is consistent. Design rule checking algorithms can be used to check consistency.

interfaces and components implement these interfaces [32]. This makes it easy to swap out one interface implementation (component) with another, say, for purposes of bug fixes, improved performance, or trying alternative implementations. Variations of interface-based programming are found in design patterns (e.g., OO decorators) and in common OO designs (e.g., frameworks) [14].

Not long ago, GenVoca was presented as example of interface-based programming.[7] The key design issue was choosing the methods of an interface. Those that were included were *fundamental* to the abstraction of that interface; those that were excluded were dismissed as not fundamental.

The problem that we and other engineers have noticed with interface-based designs is that they are brittle. We observed that the set of methods that we designated as fundamental was subjective — they were sufficient for our current needs. Over time, we longed for other methods to be included, and periodically we would indeed extend the set of methods in our interfaces. However, when new methods are added to a standardized interface, all components that export that interface had to be (manually) updated. After an extension, we would be happy for a while until we discovered a new set of methods that needed to be added, and the update cycle would repeat.

The problem with this, of course, is that we couldn't subsequently customize our interfaces or our components. It was simply too much work to eliminate unneeded groups of methods from interfaces and components. The impact of interface extensions is negative: interfaces become fat and components suffer code bloat. Other techniques have been developed to address this problem, but they too have limitations. COM, for example, requires that a new interface be published rather than changing an existing interface. While this works, it still requires a manually-introduced extension to each component that is to implement that new interface. The visitor design pattern allows almost arbitrary method extensions to existing components [14]. Access to private data members and methods of components is precluded to visitors, and this can be problematic. Also, it is useful for extensions to add new data members to components, and this too is problematic using visitors.

It is easy to recognize the concept of gluons and Origami in this situation. Each row represents either a standardized interface or a component that implements such an interface. Columns represent semantically cohesive groups of methods — features — where one column defines a "core" set of methods and other columns represent optional additional extensions to this set. Matrix (column) folding corresponds to the construction of interfaces and components that are customized for a desired set of interface extensions with their implementing components.

The need for Origami arises because abstractions change over time. Changes tend to be incremental and optional. That is, abstractions change by incremental leaps in understanding, and these leaps are needed for building specialized classes of applications. The contribution of this paper is a general model and a set of techniques that allow us to evolve both conventional components and implementations of feature refinements statically in an automatic and declaratively-specified way. Such flexibility is useful in generating software. For situations dealing with third-party components, where extensibility without recompilation is a major goal, it might not work as well. However, there is *no* requirement that feature refinements must be composed statically; they can be composed dynamically as well [31]. Unfortunately, dynamically-composable refinements are not as well-understood as statically-composable refinements.

10. Related Work

The idea that features have features is well-established in the product-line community. Feature diagrams, which are typically hierarchies of features, i.e., parent features are defined to have aggregate sets of child features, was first introduced in the FODA methodology [18] and has been improved by others [10]. Our contribution shows how the idea of features-of-features translates into product-family models based on feature refinements.

As mentioned earlier, there are other models of program development that are very similar to GenVoca, the most prominent of which are **`AspectJ`** and **`Hyper/J`**. **`AspectJ`** [1] offers two flavors of cross-cutting implementations: static and dynamic. Static cross-cuts are almost identical to GenVoca layers: they can add new data members and new methods to existing classes. Dynamic cross-cuts, where explicit pointcut-advice pairs are defined, can emulate the refinement (overriding) of methods offered by inheritance. What aspects *cannot* currently represent is the addition of new classes; in GenVoca terms, aspects only extend existing classes. (At least, we have been unable to add classes in aspect definitions that can be subsequently refined). With simple work-arounds, we have implemented GenVoca generators using **`AspectJ`**. These preliminary results suggest that compositions of layers can be modeled as compositions of aspects. Therefore, we believe that the Origami example in this paper could be implemented using **`AspectJ`** and thus our results are relevant to AOP in that they show how aspects can scale to product-families.

Admittedly, **`AspectJ`** can do more than just implement layers (modulo our comments above), and in fact, we are

[7] Which actually it still is. Layers have interfaces, although in recent papers including this one, this "feature" of layer implementation has been down-played. See [8, 22].

focussing on the least novel part of **`AspectJ`**. But it is also the case that what we and others have been able to do with GenVoca generators has never been done in AOP. Our work provides an opportunity to enhance AOP's appeal from a novel direction.

Our work is more closely related to *Multi-Dimensional Separation of Concerns (MDSC)*. MDSC is the idea that modularity relationships can be understood in terms of an n-dimensional space, called a *hyperspace*, of units [33, 23, 24]. A *unit* can be primitive (such as an individual method or variable) or compound (e.g., a class or package). Each dimension is associated with a set of similar concerns, such as a set of classes or a set of features; different values along a dimension are different members of this set (e.g., $\text{class}_1 \ldots \text{class}_n$ or $\text{feature}_1 \ldots \text{feature}_n$). A *hyperslice* is the set of units that pertain to a concern; it is an (n-1)-dimensional space where one coordinate value (e.g., a concern) is fixed. A *hypermodule* is a set of hyperslices and a set of integration relationships that dictate how the units of hyperslices are to be integrated or composed to form a program.

`Hyper/J` is the flagship tool for MDSC [34]. We have used **`Hyper/J`** to implement GenVoca product-line models. GenVoca layers have direct implementations as hyperslices, and layer compositions are hyperslice compositions. Again, we believe that the Origami model and its results are directly applicable to **`Hyper/J`**. Origami is a 2-dimensional example of MDSC, where both dimensions are features and units are gluons. Further, the strength of MDSC models is that they do not impose fixed modularization hierarchies, and this flexibility is present in Origami matrices. As with **`AspectJ`**, **`Hyper/J`** can do more than just compose hyperslices. Our contribution is that we can provide sophisticated examples of product-lines and product-families to **`Hyper/J`** researchers.

In summary, GenVoca, **`Hyper/J`** (MDSC), and **`AspectJ`** (AOP) have substantial overlaps. What distinguishes GenVoca and Origami is an algebra for organizing features into programs.

Other related work deals with tool integration [31]. Cross-cuts are problematic when new features can impact every product in a product-family. However, instead of designing a system to easily handle features, [31] explored how a product family can be designed in such a way that new features can be added by modifying a single class — a design that eliminates cross-cuts. The advantage is that it can be applied to legacy software and that it ensures that existing tools will be able to work with new additions to the program family without recompilation. The authors emphasize that in their design, the cost of evolutionary change is proportional to its apparent size in specification. The disadvantage is that this technique only applies to some features, so in fact their approach is complimentary to our own.

11. Conclusions

Features have proven their value in raising the level of abstraction in modularity in building and customizing individual programs. The question is: do features scale to larger program organizations, such as program families? We showed that they do in the context of GenVoca generators, which has not been done before. We discovered that features themselves have internal structures — features of features — which we called gluons. Gluons are arranged and composed in very regular ways, so that compositions of gluons yields both familiar and formerly "atomic" features, as well as an interesting and what we now believe is a common phenomena of facets. Facets cross-cut features and compositions of them yield fully-formed features. In essence, we have identified a new class of composition relationships among features that were not previously known.

There is anecdotal evidence that supports our work. Engineers have repeated the observation that there is something about program scale that introduces complexity one doesn't find in small programs. Our work reveals one reason: there are relationships and constraints that exist among gluons when building program families. If there is no way (or only ad hoc ways) of expressing and satisfying these constraints, it is no wonder why scaling programs introduces complexity. At least now we have a way to express and reason about such constraints. Undoubtedly there are even more relationships to be discovered.

The key to our success is *how* we represent and manipulate these relationships. Using GenVoca formulations allows us to capture these regularity relationships as matrices of functions and constants that can be folded into equations. That is, we can *reason* about software designs as equations. We explained that our results are not GenVoca-specific, in particular, how Origami has direct relationships to AOP and MDSC models. We believe Origami is important, because others will encounter it as feature refinement models scale to produce more complex systems.

Acknowledgements. We thank Jack Sarvela for pointing out the relationship of Origami to internationalization customizations of Windows programs. We also thank anonymous referees for helping us to better clarify our presentation.

12. References

[1] AspectJ. Programming Guide. **`http://aspectj.org/doc/proguide`**

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

[3] D. Batory and B.J. Geraci, “Composition Validation and Subjectivity in GenVoca Generators”, *IEEE Transactions on Software Engineering*, Feb. 1997, 67-82.

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

[5] D. Batory, G. Chen, E. Robertson, and T. Wang, “Design Wizards and Visual Programming Environments for GenVoca Generators”, *IEEE Trans. Software Engineering*, May 2000.

[6] D. Batory, C. Johnson, R. MacDonald, and D. von Heeder, “Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study”, to appear in *ACM TOSEM*.

[7] I. Baxter, “Design Maintenance Systems”, *CACM*, April 1992.

[8] R. Cardone, A. Brown, S. McDirmid, and C. Lin, “Using Mixins to Build Flexible Widgets”, *AOSD 2002*.

[9] K. Czarnecki and U.W Eisnecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[10] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker, “Generative Programming for Embedded Software: An Industrial Experience Report”, GCSE/SAIG 2002.

[11] A. van Deursen and P. Klint, “Little Languages: Little Maintenance?”, *SIGPLAN Workshop on Domain-Specific Languages*, 1997.

[12] E.W.Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[13] Flatt, M., Krishnamurthi, S., and Felleisen, M. “Classes and Mixins”. *ACM Principles of Programming Languages*, San Diego, California, 1998, 171-183.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[15] M. Griss, “Implementing Product-Line Features by Composing Component Aspects”, *First International Software Product-Line Conference*, Denver, August 2000.

[16] W. Harrison and H. Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)”, *OOPSLA 1993*, 411-427.

[17] Javadoc — The Java API Documentation Generator. Sun Microsystems, **`http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javadoc.html`**

[18] K.C. Kang, et al., Feature-Oriented Domain Analysis Feasibility Study, SEI 1990.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, *ECOOP 97*, 220-242.

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

[21] M. Mezini and K. Lieberherr, “Adaptive Plug-and-Play Components for Evolutionary Software Development”, *OOPSLA 1998*, 97-116.

[22] S. McDirmid, M. Flatt, and W.C. Hsieh, “Jiazzi: new-Age Components for Old-Fashioned Java”, *OOPSLA 2001*.

[23] H. Ossher and P. Tarr. “Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software.” *CACM* October 2001.

[24] H. Ossher and P. Tarr, “Multi-dimensional separation of concerns and the Hyperspace approach.” In *Software Architectures and Component Technology* (M. Aksit, ed.), 293-323, Kluwer, 2002.

[25] T. Reenskaug, et al., “OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems”, *Journal of Object-Oriented Programming*, 5(6): October 1992, 27-41.

[26] R. Sessions, *COM+ and the Battle for the Middle Tier*, Wiley Computer Publishing, 2000.

[27] C. Simonyi, “The Death of Computer Languages, the Birth of Intentional Programming”, *NATO Science Committee Conference*, 1995.

[28] Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”. *ECOOP*, July 1998.

[29] Y. Smaragdakis and D. Batory, “Scoping Constructs for Program Generators”. *Generative and Component-Based Software Engineering (GCSE)*, September 1999.

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

[31] K.J. Sullivan and Notkin, D., ``Reconciling Environment Integration and Software Evolution," *ACM TOSEM* July 1992.

[32] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

[33] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton, Jr., “N Degrees of Separation: Multi-Dimensional Separation of Concerns”, *ICSE 1999*.

[34] P Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM Corporation, 2001. **`http://www.research.ibm.com/hyperspace`**.

[35] M. Van Hilst and D. Notkin, “Using Role Components to Implement Collaboration-Based Designs”, *OOPSLA 1996*, 359-369.

Scaling Step-Wise Refinement

Don Batory, Jacob Neal Sarvela
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{batory, sarvela}@cs.utexas.edu

Axel Rauschmayer
Ludwig-Maximilians-Universität München
Institut für Informatik
D-80538 Munich, Germany
axel@rauschma.de

Abstract[1]

Step-wise refinement is a powerful paradigm for developing a complex program from a simple program by adding features incrementally. We present the *AHEAD (Algebraic Hierarchical Equations for Application Design)* model that shows how step-wise refinement scales to synthesize *multiple programs* and *multiple non-code representations*. AHEAD shows that software can have an elegant, hierarchical mathematical structure that is expressible as nested sets of equations. We review a tool set that supports AHEAD. As a demonstration of it's viability, we have bootstrapped AHEAD tools solely from equational specifications, generating Java and non-Java artifacts automatically, a task that was accomplished only by ad hoc means previously.

1 Introduction

Step-wise refinement is a powerful paradigm for developing a complex program from a simple program by incrementally adding details [9]. The program increments that we consider in this paper are *feature refinements* — modules that encapsulate individual features where a feature is a product characteristic that is used in distinguishing programs within a family of related programs (e.g., a product-line) [13].

There are many implementations of feature refinements, each with different names, capabilities, and limitations: layers [2], feature modules [17], meta-classes [10], collaborations [23][24], subjects [14], aspects [18], and concerns [27]. More general than traditional packages that encapsulate sets of complete classes, a feature refinement can also encapsulate *fragments* of multiple classes. Figure 1 depicts a package of three classes, `c1`—`c3`. Refinement `r1` *cross-cuts* these classes, i.e., it encapsulates fragments of `c1`—`c3`. The same holds for refinements `r2` and `r3`. Composing refinements `r1`—`r3` yields a package of fully-formed classes `c1`—`c3`. Because refinements reify levels of abstraction, feature refinements are often called *layers* — a name that is visually reinforced by their vertical stratification of `c1`—`c3` in Figure 1. As the concepts of refinements, layers, and features are so closely related, we use their terms interchangeably. In general, feature refinements are modular, albeit unconventional, building blocks of programs.

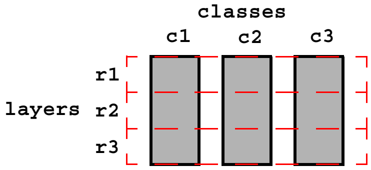


Figure 1. Classes and Refinements (Layers)

Tools that synthesize applications by composing feature refinements are *generators* whose focus has been on the production of source code for individual programs. This is too limited. Today's systems are not individual programs, but rather groups of different programs collaborating in sophisticated ways. Client-server architectures are examples, and so are tool suites, such as MS-Office. Further, systems are not solely described by source code. Architects routinely use different knowledge representations (e.g., process models, UML models, makefiles, design documents) to capture an application's design [16]. Each representation encodes different design information and is expressed in its own *domain-specific language (DSL)*, such as a UML diagram, OCL, XML, etc.

The contribution of this paper shows how step-wise refinement scales to the simultaneous synthesis of multiple programs and multiple non-code representations written in different DSLs. The challenge is not one of possibility, as ad hoc ways are used now. Rather, the challenge is to create a mathematical model of application synthesis that treats all representations — code and non-code, individual programs and multiple programs — in a uniform way. By expressing the refinement of representations as equations, we not only simplify tool development (as equations are ideal for program manipulation) but also lay the groundwork for specifying, generating and optimizing application designs of considerable complexity using simple algebraic techniques.

We begin with a review of the GenVoca model, which shows how the *code* representation of an *individual* program is expressed by an equation. We then present the *Algebraic Hierarchical Equations for Application Design (AHEAD)* model that generalizes equational specifications to multiple

[1] This work was supported in part by the U.S. *Army Simulation and Training Command (STRICOM)* contract N61339-99-D-10 and *Deutsche Forschungsgemeinschaft (DFG)* project WI 841/6-1 "InOpSys".

programs and multiple representations. AHEAD is related to other models, such as Aspect-Oriented Programming [18] and Multi-Dimensional Separation of Concerns [22], and thus our results are not GenVoca-specific. We review a tool set that supports AHEAD. As a demonstration of AHEAD's viability, we have bootstrapped AHEAD tools, generating over 100K LOC of Java (and other non-code artifacts) solely from equational specifications automatically, a task that was accomplished only by ad hoc means previously. We believe that this is the most sophistated system automatically built using step-wise refinement.

2 GenVoca

GenVoca is a design methodology for creating application families and architecturally-extensible software, i.e., software that is customizable via module additions and removals [2]. It follows traditional step-wise refinement [9] with one major difference: instead of composing thousands of microscopic program refinements (e.g., `x+1⇒inc(x)`) to yield admittedly small programs, GenVoca scales refinements so that each adds a feature to a program, and composing few refinements yields an entire program.

2.1 Model Concepts

Programs are *values* and refinements are *functions* that add features to programs. Consider the following constants that represent programs with different features:

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

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

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

A multi-featured application is an *equation* that is a named expression. Different equations define a family of applications, such as:

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

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

Note that a function represents both a feature *and* its implementation — there can be different functions with different implementations of the *same* feature:

```
k1(x)    // adds k with implementation1 to x
k2(x)    // adds k with implementation2 to x
```

When an application requires the use of feature **k**, it is a problem of *equation optimization* to determine which implementation of **k** is best (e.g., provides the best performance)[2]. It is possible to automatically design software (i.e., produce an equation that optimizes some criteria) given a set of declarative constraints for a target application. An example of this kind of automated reasoning — historically called *automatic programming* [1] — is [5].

Although GenVoca constants and functions seem untyped, typing constraints do exist as design rules. *Design rules* capture syntactic and semantic constraints that govern legal compositions. It is common that the selection of a feature will disable or enable the selection of other features. The details of design rules are not relevant to our paper and can be found in [3].

2.2 Model Implementation

A GenVoca constant is a set of classes. Figure 2 depicts a constant **i** that encapsulates four classes (a_i—d_i). A GenVoca function is a set of classes and class refinements. A *class refinement* adds new data members, methods, and extends or overrides existing methods of a target class. It could be implemented as a subclass of the target class, *with the provision that the subclass assumes the name of the target class*. In this sense, class refinement is different from typical subclassing in that true subclasses have names that are distinct from their parent class.

Figure 2 shows the result of applying function **j** to **i**: three classes are refined and another class is added, where the vertical lines in Figure 2 denote class refinement. That is, **j** encapsulates a cross-cut that refines classes a_j, c_j, and d_j, and adds class e_j. Figure 2 also shows the application of function **k** to **j(i)**, resulting in the refinement of two classes. In general, a forest of inheritance hierarchies is created as layers are composed, and this forest grows progressively broader and deeper as the number of features increase [6].

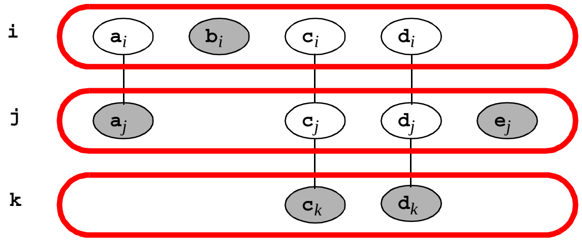


Figure 2. Implementing Layers as Collaborations

[2] Different equations represent different programs and equation optimization is over the space of semantically equivalent programs. This is identical to relational query optimization: a query is represented by a relational algebra expression, and this expression is optimized. Each expression represents a different, but semantically equivalent, query-evaluation program.

Linear inheritance chains, called *refinement chains*, are common in this method of implementation. The rule is that only the bottom-most class of a chain is instantiated, because this class implements all capabilities that were assigned to it. These classes are shaded in Figure 2. For example, the bottom-most class of the `c` refinement chain implements the capabilities defined in $\mathtt{c}_i$, $\mathtt{c}_j$, and $\mathtt{c}_k$ from features `i`, `j`, and `k` respectively. Stated another way, $\mathtt{c}_i$ is a class (constant) and $\mathtt{c}_j$ and $\mathtt{c}_k$ are class refinements (functions). The class `c` that is produced in Figure 2 can be expressed equationally as $\mathtt{c} = \mathtt{c}_k(\mathtt{c}_j(\mathtt{c}_i))$.

The simplest implementations of feature refinements are templates [24][29]. More sophisticated ways include generators [5][20], program transformations [8], and objects [26]. Please consult these references for further details.

3 Scaling Refinements

Four ideas have led us to a major generalization of the GenVoca model. First, a program has many different representations besides source code. A program can be defined by UML documents, process diagrams, makefiles, performance models, design rule files, etc. in addition to code, each written in its own DSL.

Second, conventional notions of modularity must be broadened: *a module is a containment hierarchy of artifacts*. For example, an object-oriented class is a module; it encapsulates a set of data members and methods. A package is another kind of module; it encapsulates a set of `.java` and `.class` files. A J2EE EAR file is also a module; it encapsulates Java JAR files, deployment descriptors, and web archive files (e.g., HTML files). A program is yet another kind of module; its representation is expressible as a containment hierarchy of artifacts — code, makefiles, etc.

Third, the impact of refinements need not be limited to source code. When a program is refined, any or all of its representations may be changed. This is required so that all of its artifacts (code files, design rules, etc.) remain consistent. Thus if a program is represented by a containment hierarchy, a program refinement is a function that transforms containment hierarchies. That is, a refinement may alter a containment hierarchy by adding new nodes (e.g., a refined program may have new `.java` or HTML files) and refining existing nodes (e.g., existing `.java` and HTML files are updated).

Fourth, refining non-code artifacts is intuitively evident: you start with an original document and you produce an updated document. But what does refinement mean? And what general principle should guide refinement? An example reveals a basic strategy that we have found to be practical. A makefile is a typical non-code artifact. Figure 3a shows a makefile with three targets: `main`, `common`, and `clean`. `common` must be built before `main`; `clean` has no dependencies. Suppose these targets are part of a `base` feature. Figure 3b shows the refinement of `base` by `foo`, which encapsulates a cross-cut of targets that adds a file `D` to compile in `main` and file `E` to compile in `common`. Figure 3b shows a further refinement of `foo(base)` by `bar`, which adds another file `Q` to compile to `main` and a new instruction (`delete *.ser`) to `clean`. From this example, the idea of refining a makefile is clear.

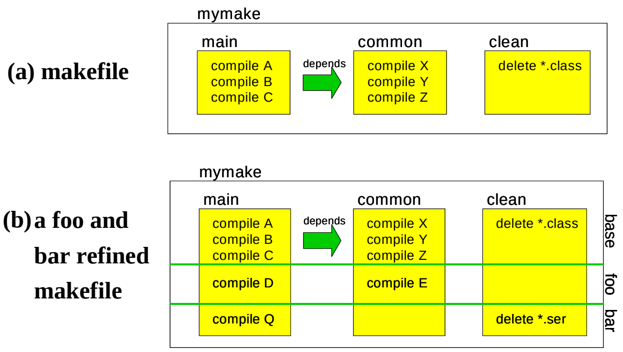


Figure 3. Makefiles and their Refinements

Figure 3 reveals a makefile to have a class structure. That is, a makefile is analogous to a class with methods as targets. Refinement of makefiles refines targets and may add new targets, properties, etc. This example illustrates the *Principle of Uniformity*: treat all non-code artifacts as classes, and refine them in an analogous manner as code.

Interestingly, most artifact types in use today (XML, HTML, MIF, etc.) have or can have a class structure and thus are *object-based*. However, these types are rarely *object-oriented* — few allow inheritance or more general refinement relationships to be defined among its files. There is, for example, no support for inheritance relationships to relate different XML documents. For representations like XML, adding inheritance or refinement relationships seems reasonable as a plethora of tools for manipulating XML documents are available and can be used to define such relationships. However, other representations — MS Word for example — are so complex that defining inheritance relationships among Word documents seems daunting. Never-the-less, the idea is clear: the only thing to be done when introducing a new artifact type is define what refinement means.[3] Given the ability to refine non-code artifacts, we are in the position to scale refinements in a significant way.

[3] Because it is possible to derive artifacts (e.g., `.class` files are derived from `.java` files via `javac`), composition operators are needed only for the most basic artifact types. Maintaining the consistency among "basic" types in a refinement (e.g., English explanations of source code) is something that is accomplished manually.

4 AHEAD

GenVoca expressed the *code* representation of an *individual* program as an equation. The model that we present here, called *Algebraic Hierarchical Equations for Application Design (AHEAD)*, expresses an arbitrary number of programs and representations as nested sets of equations, a form that is ideal for generators to manipulate. In this section, we show how AHEAD constants, functions, and their compositions are represented, and illustrate the power of the model. In Section 5, we review a tool set for AHEAD.

4.1 Constants and Functions

Base artifacts are *constants* and artifact refinements are *functions*. An artifact that results from a refinement chain is modeled equationally as a series of functions (refinements) applied to a constant (base artifact).

Figure 4a depicts our graphical notation for a GenVoca constant f that encapsulates base artifacts (henceforth called *units*) a_f—c_f. Instead of pictures, we express constant f mathematically as a set of constants: $f = \{ a_f, b_f, c_f \}$. Similarly, Figure 4b depicts our graphical notation for a GenVoca function h that encapsulates functions a_h and b_h and constant d_h. Function h can be expressed mathematically as a set of functions and constants: $h = \{ a_h, b_h, d_h \}$.

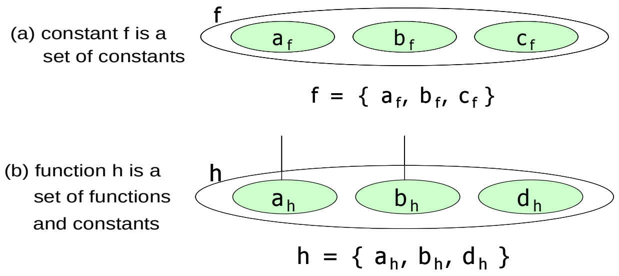

(a) constant f is a set of constants
f
a_f
b_f
c_f
$f = \{ a_f, b_f, c_f \}$
(b) function h is a set of functions and constants
h
a_h
b_h
d_h
$h = \{ a_h, b_h, d_h \}$

Figure 4. Constants and Functions as Sets (Collectives)

We use the terms *model* or *collective* as an alternative to "set". We say models and collectives have units which are constants or functions.

4.2 Composition

Instead of writing h(f) to denote the composition of h with f, we henceforth write $h \bullet f$. The composition of collectives is governed by the rules of inheritance. Namely, all units of the parent (inner or right-hand-side) collective are inherited by the child (outer or left-hand-side) collective. Further, units with the same name (ignoring subscripts) are composed pairwise with the parent unit as the innermost term:

$$
\begin{aligned}
h \bullet f &= \{ a_h, b_h, c_h \} \bullet \{ a_f, b_f, d_f \} \\
&= \{ a_h \bullet a_f,\ b_h \bullet b_f,\ c_h,\ d_f \}
\end{aligned}
$$

Equivalently, $h \bullet f$ is a set of equations where equation names are unit names without subscripts:

$$
\begin{aligned}
h \bullet f &= \{ a, b, c, d \} \\
\text{where } a &= a_h \bullet a_f \\
b &= b_h \bullet b_f \\
c &= c_h \\
d &= d_f
\end{aligned}
$$

Each expression in a collective defines a refinement chain of an artifact that is to be produced. Figure 5 shows this correspondence between our graphical notation and its AHEAD expression: $a_h \bullet a_f$ is the refinement chain for artifact a and $b_h \bullet b_f$ is the chain for b. Artifacts c and d are unrefined from their original definitions.

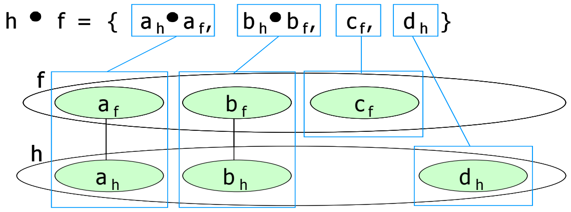

$h \bullet f = \{ a_h \bullet a_f, b_h \bullet b_f, c_f, d_h \}$
f
h
a_f
b_f
c_f
a_h
b_h
d_h

Figure 5. Expression and Refinement Chain Correspondence

Containment Hierarchies. Compound artifacts are expressed by recursion: units may themselves be collectives. Composition of compound artifacts is achieved by recursively composing collectives. For example, suppose a_h and a_f in Figure 5 are the collectives $a_h = \{ x_h, z_h \}$ and $a_f = \{ x_f, y_f \}$. The expression $a_h \bullet a_f$ expands to the collective $\{ x_h \bullet x_f, y_f, z_h \}$. The depth to which collectives are nested is the *rank* of the collective. {} is an empty collective of rank 0; {{}} is a collective of rank 1, and so on.

A compound artifact is a containment hierarchy. A Java program p, for example, is a compound artifact and thus is a non-leaf node; its leaves might be the set of Java files {x.java, y.java, ...} that implement it, the set of HTML files {x.html, y.html, ...} that JavaDoc produces from these Java files, the set of .class files {x.class, y.class, ...} produced by javac, and so on.

Thus, a feature — whether it represents an AHEAD constant or a function — is defined by a tree of artifacts. When features are composed, all of their corresponding artifacts are composed. Thus, feature composition has a simple interpretation.

Polymorphism. The composition operator $\bullet$ is polymorphic. Artifacts are composed by operators that are specific to that artifact type. Java files, design rule files, XML files, etc. will each have their own unique implementation of the $\bullet$ operator. Thus, if there are *n* different artifact types, there will be *n* different artifact-type-specific composition operators. (As we will see in Section 5, there may be several composition operators for a given artifact type).

Further, the operator for composing collectives of rank *n* is no different than the operator for composing collectives of rank *n+1*, for *n≥0*. We call this the *Principle of Abstraction Uniformity*, which is a special case of the Principle of Uniformity. Imposing uniformity on all levels of abstraction produces a very compact and powerful algebra for defining and composing systems.

Scalability. OO programming languages that support parameterized inheritance can define and refine code artifacts, but are unsuitable for non-code representations. In contrast, equations elegantly express refinement relationships for *all* representations. Furthermore, *equations enable step-wise refinement and its generators to scale*. Instead of building one huge generator that deals with all possible program representations (which itself is impractical), it is *much* easier to build an elementary tool — here called a `composer` — that expands a high-level equation into its constituent artifact equations. The `composer` can then submit its generated equations to artifact-type-specific composition tools to generate the artifacts of the target system. Thus, a simple `composer` tool does the work of orchestrating other relatively simple tools to produce the complex set of artifacts that comprise a synthesized system.

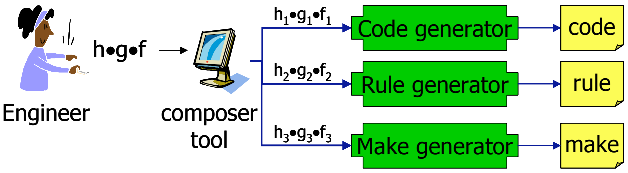


Figure 6. Organization of AHEAD Generators

4.3 Metamodels

One way in which to gauge the power of a model is its ability to express itself. A *metamodel* is a model whose instances are models. The UML metamodel is a classical example: it is a UML model whose instances are other UML models. AHEAD metamodels are collectives of models. Consider model `M` that contains three units `a`—`c`:

```
M = { a, b, c }
```

Metamodel `MM` contains three units `AAA`—`CCC`, each of which is a collective with a single unit:

```
MM = { AAA, BBB, CCC }
   = { {a}, {b}, {c} }
```

Model `M` is defined by an equation using metamodel `MM`:

```
M = AAA ● BBB ● CCC
```

where composition creates `M` by disjoint union, since none of the units share names. The following are interesting uses of metamodels.

Service Packs. A *service pack* is an update of model. A *service pack metamodel* `SP` contains an initial model M_0 and a series of service pack updates S_1, S_2, S_3, each of which incrementally updates a model:

$$SP = \{ M_0, S_1, S_2, S_3, \ldots \}$$

A special composition operator ♦, called *replace*, is used to apply the changes of a service pack to an existing model. If U_k and U_j are primitive (i.e., non-compound) units, the law of the replace operator is:

$$U_k \blacklozenge U_j = U_k \qquad \texttt{// } U_k \texttt{ replaces } U_j$$

Otherwise ♦ is identical to the composition ● operator for collectives. (♦ is actually a special case of ●). Thus, a model `M` that is up-to-date w.r.t. service pack S_3 is defined by the equation:

$$M = S_3 \blacklozenge S_2 \blacklozenge S_1 \blacklozenge M_0$$

That is, the effects of S_1 are applied to M_0 by replacing old base artifacts with new ones, and adding new artifacts. The same for S_2 and S_3. Special primitive units might be used to indicate the physical deletion (rather than replacement) of designated files.[4]

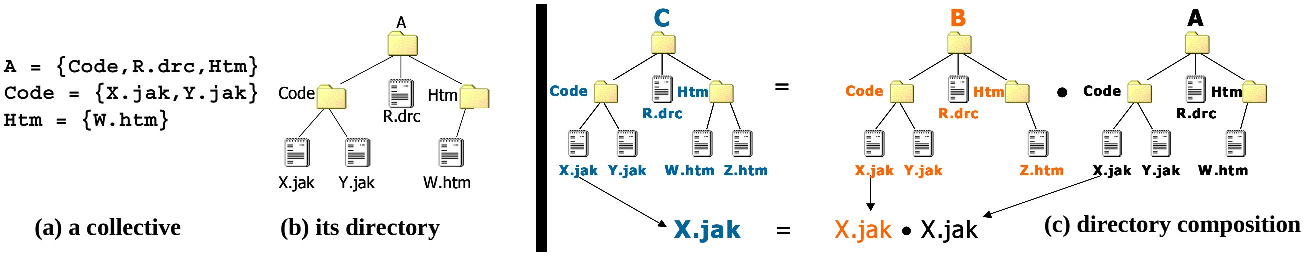
Origami. A much more sophisticated use for AHEAD metamodels is generating equations for AHEAD tools from a single AHEAD metamodel equation. See [7].

5 AHEAD Tool Support

The simplest way to implement a collective is as a file system directory. A directory's contents are either files (primitive units) or subdirectories (collectives). Figure 7a shows a collective that defines layer `A`; it consists of a unit, `R.drc` (a design rule file), and two collectives, `Code = {x.jak, y.jak}` (`.jak` files are extended-Java files) and `Htm = {W.htm}`. Figure 7b depicts its representation as a directory.

Feature composition is directory composition. A composite directory is produced whose organization is isomorphic to the directories from which it was composed. Figure 7c shows a composition of features `A` and `B`, the result is feature `C`. Corresponding units in each directory are composed to produce a compound unit. For example, `x.jak` of `C` equals `x.jak` of `B` composed with `x.jak` of `A`.

[4] A design rule is: S_i must be applied before S_{i+1} for i≥0.



(a) a collective

(b) its directory

(c) directory composition

Figure 7. Collectives as Directories

Given this organization, we have built many tools to support AHEAD using the *Jakarta Tool Suite (JTS)* [4]. JTS is a set of compiler-compiler tools that use GenVoca models to build *product-lines* (i.e., families) of preprocessors for extended Java languages. Almost all AHEAD tools were built using JTS, and were written in a JTS-produced dialect of Java that includes refinement declarations, meta-programming constructs (e.g., Lisp "quote" and "unquote"), and hygienic macros. In Section 6 we describe how we recently bootstrapped our implementation where JTS is no longer used and AHEAD tools are now creating AHEAD tools.

The main tool of AHEAD is the `composer`, which takes an equation as a command-line input, and recursively expands the equation into its nested collective equivalent. It then creates a composite feature directory (whose name is that of the input equation), and invokes artifact-specific composition tools to synthesize artifact files from generated nested equations. `composer` itself is fairly simple, written in 4K lines of Java.

Other tools implement the composition operator ● for specific artifact types. The first tools that we built composed code artifacts, as verifying the code synthesis capabilities of AHEAD was our first priority. Subsequently, composition tools for HTML files, makefiles, equation files (files whose contents define a single AHEAD equation), design rule files, and BNF-grammar files (for synthesizing extensible preprocessors) were constructed. We anticipate the set of artifact-composition tools to grow much larger over time.

Because of space limitations, we review how AHEAD tools compose code and equation representations. Other tools and representations — such as makefiles and design rules — are described in [28].

5.1 Code Artifacts and Tools

Code files that are composed by AHEAD tools are not pure Java, but rather a superset of Java called `.jak` (pronounced "jack", short for Jakarta): this is Java extended with embedded DSLs for refinements, state machines, and meta-programming. (Remember: AHEAD tools are Java language extensible, so AHEAD can support many Java dialects). `.jak`-specific tools are invoked to compose `.jak` files. One of two different implementations of the ● operator can be used: `jampack` or `mixin`. Both take an equation as input, which defines the refinement chain of a `.jak` artifact, and produce a single, composite `.jak` file as output. A third tool, `jak2java`, translates a `.jak` file to its `.java` counterpart. Thus our two-step paradigm uses `jampack` or `mixin` to compose `.jak` files, and `jak2java` to derive the corresponding `.java` file from its composite `.jak` file (Figure 8).

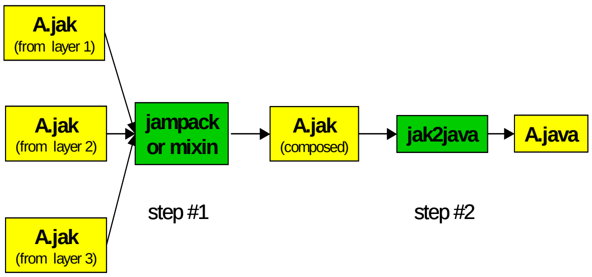


Figure 8. Composing and Translating .jak files

Source Code. A `.jak` file defines a code constant or function. A code constant is a single interface, class, or state machine. A `.jak` interface and class declaration are indistinguishable from their Java counterparts, except for a layer declaration which specifies the name of the layer to which the file belongs (see Figure 9a). More interesting is a state machine declaration, which consists of state and edge (transition) declarations. The state machine `fsm` of Figure 9b declares three states (`s1`—`s3`) and two edges (`e1`—`e2`). See [6] for more details.

A code function refines an interface, class, or state machine. The `refines` modifier distinguishes constant declarations from functions. A refinement of class `k` in Figure 9a is shown in Figure 9c. It adds a new data member (`counter2`) and a new method (`method2`). Methods can be refined/extended in the usual way by overriding their superclass method and invoking that method in the extension body. A refinement of the `fsm` state machine is shown in Figure 9d. It adds another state `s4` and edge `e3` declaration.

Composition. `jampack` and `mixin` are sophisticated tools, both over 30K Java LOC in size. Their basic functionality, however, is fairly simple. `jampack` compresses refinement chains into a single interface, class, or state machine speci-

```
layer A;

import java.util.*;

class k {
   int counter;

   int getCounter()
   {...}
}
```

(a) class constant

```
layer A;

import java.util;

State_machine fsm {
   States s1, s2, s3;

   Edge e1: s1 -> s2 ...;
   Edge e2: s2 -> s3 ...;
}
```

(b) state machine constant

```
layer B;

import foo.bar;

refines class k {
   int counter2;

   void method2()
   {...}
}
```

(c) class function

```
layer B;

import foo.bar;

refines State_machine fsm {
   States s4;

   Edge e3: s3 -> s4 ...;
}
```

(d) state machine function

Figure 9. AHEAD Code Constants and Functions

fication. The `jampack` result of composing the `k` function with the `k` constant of Figure 9 is shown in Figure 10a; the resulting class exposes the union of package imports, data members, and methods that are visible to the bottom-most class of its refinement chain. The `jampack` result of composing the `fsm` function with the `fsm` constant of Figure 9 is shown in Figure 10b. The resulting state machine exposes the union of package imports, states, and edges, that are visible to the bottom-most state machine of its refinement chain.

```
layer C;

import java.util.*;
import foo.bar

class k {
   int counter;
   int counter2;

   int getCounter()
   {...}
   int method2()
   {...}
}
```

(a)

```
layer C;

import java.util.*;
import foo.bar;

State_machine fsm {
   States s1, s2, s3;
   States s4;

   Edge e1: s1 -> s2 ...;
   Edge e2: s2 -> s3 ...;
   Edge e3: s3 -> s4 ...;
}
```

(b)

Figure 10. `jampack` Compositions of .jak Files

`jampack` was our first tool to compose code. We soon discovered that `jampack` might not be the preferred tool. To see why, a typical debugging cycle is to (a) compose `.jak` files, (b) translate the composite `.jak` file to its `.java` counterpart, (c) compile and debug the `.java` file, and (d) update the composite `.jak` file with bug fixes. This translate-compile-update cycle continues until no further changes are needed. Here lies the problem: `jampack` does not preserve layer boundaries, thus changes made to the composite `.jak` file must be manually propagated back to the original `.jak` layer files. This can be tedious and error prone.

`mixin` was created as an alternative implementation of the `.jak` composition operator. It preserves layer boundaries by creating within a single `.jak` file an inheritance hierarchy — refinement chain — of class/interface/state machine declarations, where the bottom-most declaration is `public` and all others are `abstract`. Every declaration is prefaced by a `SoUrCe` statement that identifies both the layer name and path to the file from which that instantiated declaration originated. (The case alteration in “`SoUrCe`” minimizes the likelihood of conflicting with typical program identifiers). Figure 11a-b shows the `mixin` output for the corresponding `jampack` compositions of Figure 10a-b. With the availability of `SoUrCe` statements, we have created a fourth tool, called `unmixin`, which takes a `mixin`-produced `.jak` file as input, and automatically propagates updates (comments as well as source statements) to the original layer files. Early experiences with `unmixin` suggest it can save a considerable amount of time during a development cycle.

There is yet another strategy: it is possible to edit the original layer files and recompose. Our experience suggests that debugging layers is similar to debugging templates: one instantiates a template to debug and develop it, and changes are eventually back-propagated to the original definition. We do not yet know which tools and processes engineers will prefer when developing software with AHEAD; this is a subject of a future report.

```
layer C;

import java.util.*;
import foo.bar;

SoUrCe A “A/k.jak”;
abstract class k001 {
   int counter;
   int getCounter()
   {...}
}

SoUrCe B “B/k.jak”;
public class k
   extends k001 {
   int counter2;
   void method2()
   {...}
}
```

(a)

```
layer C;

import java.util.*;
import foo.bar;

SoUrCe A “A/fsm.jak”;
abstract State_machine
   fsm001 {
   States s1, s2, s3;
   Edge e1: s1 -> s2 ...;
   Edge e2: s2 -> s3 ...;
}

SoUrCe B “B/fsm.jak”;
public State_machine fsm
   extends fsm001 {
   States s4;
   Edge e3: s3 -> s4 ...;
}
```

(b)

Figure 11. `Mixin` Compositions of .jak Files

5.2 Equation Artifacts

An *equation file* encodes an AHEAD equation. The file for `X = F●E●D` is depicted in Figure 12a; the name of the file is `X.equation` and the file itself is a text file that lists one layer per line, inner-most layer first. A refinement of an equation follows the principle of uniformity: we treat an equation as a method and a refinement is an equation that may reference a special layer called `super`, which refers to the parent equation. A refinement of an equation is depicted in Figure 12b, and the result of composing the equation files of Figure 12a-b is Figure 12c. If a refinement does not reference `super`, it is a constant and overrides the parent definition. Depending on the replacement of `super` in an equation file, a refinement might add new layers before, after, or around the original equation file (like before, after, and around methods).

```
                                C
                                D
D               C               E
E               super           F
F               G               G
```

(a) constant (b) function (c) composition

Figure 12. Equation Files

Equation files are useful as command-line input to `composer`, and for implementing metamodels [7].

6 Applications

AHEAD is being used to build next-generation distributed *fire support simulators (FSATS)* for the U.S. Army *Simulation, Training, and Instrumentation Command (STRICOM)*. Several years ago, we built a layered prototype of FSATS [6]. As the first non-trivial test of AHEAD, we converted the prototype's source code, which included classes, interfaces, and state machines, into AHEAD layers. We also added design rule files and makefiles, so when AHEAD layers are composed, three very different representations of FSATS (code, design rules, makefiles) were synthesized. The prototype was defined by a single AHEAD equation composing 21 layers, yielding 90 files and 4500 Java LOC.

The next significant test was bootstrapping AHEAD itself. As mentioned earlier, AHEAD tools were initially built using JTS. To bootstrap AHEAD, we converted JTS source into AHEAD layers. In addition to code representations, AHEAD directories included grammar (BNF) files, which defined the syntax of optional Java language extensions. We used Origami, an AHEAD metamodel, to generate equations for AHEAD tools, including `jampack`, `mixin`, `unmixin`, and `jak2java`, and then used AHEAD tools to synthesize their executables [7]. To convey the complexity of this bootstrapping step, there are 69 distinct AHEAD layers constituting a code base of 33K `.jak` LOC. An Origami-generated equation references approximately 23 layers, where about 10 layers are shared among AHEAD tools. Each tool is generated by composing the `.jak` and grammar representations of each layer, and translating their representations to Java. Each AHEAD tool is sizable, having over 30K Java LOC. Thus using only equational specifications, we are generating the AHEAD tool suite whose complexity exceeds 100K Java LOC [7] *automatically*, a task that was accomplished only by ad hoc means previously. We believe this is the most sophisticated system ever produced automatically by step-wise refinement.

The real test of AHEAD will come with the redesign and implementation of the next generation of FSATS. This and the experience of using AHEAD will be the subject of future papers.

7 Future Work

AHEAD raises many interesting questions including:

- The ability to analyze designs using tools such as model checkers will be critical to future design technologies. How do such tools fit with refinements and AHEAD designs? Preliminary results are encouraging — see [19].
- There are *many* operators on collectives and units, besides the ● and ♦ operators. `javac`, for example, could be an operator on a collective that compiles the `.java` files in a collective. `javaDoc` could be an operator on a collective that generates HTML documentation from the `.java` files in a collective. By equating standard tools with operators on collectives, we have an algebra for software development. Once software is specified equationally, it is amenable to automated optimization and reasoning.
- Many refinements that impact — cross-cut — different parts of programs require more sophisticated implementations of refinements than used in AHEAD. For example, information from multiple artifacts may be used to decide how to refine other artifacts en mass. How can such functions be modeled and implemented?
- AHEAD constants and functions are typed. The layer directories that we have composed with AHEAD have the same structure (or type), but our tools are not yet sophisticated enough to validate this assumption. A theory is needed to type refinements and artifact hierarchies.
- AHEAD has been developed with functions that have at most one parameter. There are GenVoca models with functions with multiple parameters [2]. How can such functions be implemented in AHEAD? Why are functions with one parameter so common? We suspect that

the answer is not how domains are represented, but rather our (implicit) use of currying which expresses a function of multiple parameters as a composition of functions with single parameters.

8 Related Work

Among the most advanced work on generators is that of *Model Integrated Computing (MIC)* at Vanderbilt [25]. MIC embraces the concept that architects use multiple representations to specify application designs, and MIC generators have been developed to synthesize *graphical domain-specific languages (GDSLs)* that architects can use to specify their designs. Information that is collected from GDSL specifications are integrated and stored in a database. Specific artifacts of a design, ranging from source code to representations for analysis tools (e.g., model checkers), can be extracted and deduced from this database.

MIC has had great success in synthesizing software and hardware in engineering and manufacturing domains where the building blocks of systems and their composition-by-construction paradigms are well-understood. Where MIC has had less success is in areas of classical software applications where the building blocks and construction-by-composition paradigms are not well understood. We believe this is where AHEAD contributes: AHEAD shows how domains can be modularized as features and how applications of the domain are composed from them.

Domain-specific languages have long been recognized as a more efficient way in which to (a) specify applications and (b) integrate domain-specific analyses to validate DSL programs [30]. DSL usage is increasing, and has been particularly successful in the specification of product-lines [31]. AHEAD not only embraces the use of DSLs as the primary means for specifying artifacts, but also advocates that DSL programs can be refined. (Our refinement of state machines and equation files in AHEAD are prime examples; state machines are expressed in a DSL embedded in Java; equation files are written in a stand-alone DSL). It is this scaling of refinements to code and non-code artifacts that is a distinguishing feature of AHEAD.

Aspect Oriented Programming (AOP) is a program refinement technology [18]. AHEAD essentially uses templates to express refinements. A more sophisticated way is to use special compilers to implement AHEAD functions that perform computations on a collective to determine how that collective is to be modified (or “advised”). Aspects are specifications of refinements, and aspect weavers execute these specifications on input programs. Thus, aspects provide another important kind of AHEAD function that is currently lacking in the AHEAD tool set. Gray has shown how aspects apply to non-code artifacts [12]. AHEAD shows how both code and non-code artifacts can be refined simultaneously in collection hierarchies.

Multi-Dimensional Separation of Concerns (MDSC) is another program refinement technology [22]. We have built GenVoca generators using `Hyper/J`. Layers correspond to hyperslices, and GenVoca equations correspond to compositions of hyperslices. Further, MDSC advocates that the techniques for assembling customized code from hyperslice compositions should also work for non-code artifacts as well. This conjecture inspired work on AHEAD. The contribution of AHEAD is a simple algebraic model that supports the MDSC thesis.

9 Conclusions

The future of software engineering lies in automation. Integral to this vision is the transformation of application design from an art into a science — a systematized body of knowledge that is organized around principles, ideally expressible as mathematics. As long as application design remains an art, our abilities to automate key tasks of application design will be limited.

Generators are critical to this vision. As application complexity increases, the burdens placed on generators and their ability to synthesize multiple programs and multiple representations increases. The challenge in scaling refinement-based generators is not one of *possibility*, as there are any number of ad hoc ways in which this can be done. Rather, the challenge is to show how scaling can be accomplished in a *principled* manner, so that generators are not just ad hoc collections of tools and an incomprehensible patchwork of techniques. The significance of this point is clear: generators are a technological statement that the development of software in a domain is understood well enough to be automated. However, we must make the same claim for generators themselves: the complexity of generators must also be controlled and must remain low as application complexity scales, otherwise generator technology will unlikely have wide-spread adoption.

We have presented the AHEAD model, which offers a practical solution to the above problem. The key ideas are (1) to represent the plethora of representations that define a program — both code and non-code — as a containment hierarchy, and to treat containment hierarchies as values. And (2) express feature refinements as functions that transform these values; such refinements encapsulate all the changes that are to be made to the representations of a program when a feature is added.

Doing this, we discovered that application designs have an elegant hierarchical mathematical structure that is expressed by nested sets of equations. By imposing uniformity, we (1) eliminate ad hoc complexity as containment hierarchies scale, (2) enable a small number of operators to be used to manipulate AHEAD concepts, and most importantly (3) keep generators based on step-wise refinement simple as the systems they synthesize scale in complexity.

We reviewed an implementation of AHEAD and described our first non-trivial systems constructed by its principles (FSATS and AHEAD tools). We are now in the process of building a next-generation simulator using AHEAD for the U.S. Army STRICOM and will have much more to report in the future. We believe AHEAD takes us an important step closer to realizing a broader vision of automation in software development.

Acknowledgements. We thank Melanie Kail and Mark Esslinger for their contributions to the design of AHEAD tools, and Jim Browne, Stan Jarzabek, Dewayne Perry, and Roberto Lopez-Herrejon for their helpful comments.

10 References

[1] R. Balzer, "A Fifteen-Year Perspective on Automatic Programming", in *Software Reusability II*, T.J. Biggerstaff and A.J. Perlis, Eds., Addison-Wesley, 1989.

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

[3] D. Batory and B.J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, Feb. 1997, 67-82.

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

[5] D. Batory, et al., "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE Trans. Software Engineering*, May 2000, 441-452.

[6] D. Batory, et al., "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study", *ACM TOSEM*, April 2002.

[7] D. Batory, R.E. Lopez-Herrejon, J-P. Martin, "Generating Product-Lines of Product-Families", *Automated Software Engineering, September 2002*.

[8] I. Baxter, "Design Maintenance Systems", *CACM*, April 1992.

[9] E.W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[10] I. Forman and S. Danforth, *Putting Metaclasses to Work*, Addison-Wesley, 1999.

[11] M. Flatt, S. Krishnamurthi, and M. Felleisen, "Classes and Mixins". *ACM Principles of Programming Languages*, San Diego, California, 1998, 171-183.

[12] J. Gray, et al. "Handling Crosscutting Constraints in Domain-Specific Modeling", *CACM* October 2001.

[13] M. Griss, "Implementing Product-Line Features by Composing Component Aspects", *First International Software Product-Line Conference*, Denver, August 2000.

[14] W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA 1993*, 411-427.

[15] W. Harrison, C. Barton, M. Raghavachari, "Mapping UML Designs to Java", OOPSLA 2000.

[16] A. Hein, M. Schlick, R. Vinga-Martins, "Applying Feature Models in Industrial Settings", *Software Product Line Conference (SPLC1)*, August 2000.

[17] K.C. Kang, et al., Feature-Oriented Domain Analysis Feasibility Study, SEI 1990.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", *ECOOP 97*, 220-242.

[19] H. Li, S. Krishnamurthi, and K. Fisler, "Interfaces for Modular Feature Verification", *Conf. Automated Software Engineering*, 2002.

[20] M. Mezini and K. Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development", *OOPSLA 1998*, 97-116.

[21] S. McDirmid, M. Flatt, and W.C. Hsieh, "Jiazzi: new-Age Components for Old-Fashioned Java", *OOPSLA 2001*.

[22] H. Ossher and P. Tarr. "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software." CACM 44(10): 43-50, *October 2001*.

[23] T. Reenskaug, et al., "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems", *Jour. OO Programming*, 5(6): October 1992, 27-41.

[24] Y. Smaragdakis and D. Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs", to appear *ACM TOSEM*.

[25] J. Sztipanovits and G. Karsai, "Generative Programming for Embedded Systems", *Generative Programming and Component-Based Engineering (GPCE)*, Pittsburgh, October 2002.

[26] K.J. Sullivan, and D. Notkin, "Reconciling Environment Integration and Software Evolution", *ACM TOSEM*. 1, 3, July, 1992, 229-268.

[27] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton, Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *ICSE 1999*.

[28] University of Texas Center for Agile Technology, "AHEAD Tool Documentation", 2002.

[29] M. Van Hilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs", *OOPSLA 1996*, 359-369.

[30] A.van Deursen and P. Klint, "Little Languages: Little Maintenance?", *SIGPLAN Workshop on Domain-Spec. Lang.*, 1997.

[31] D. Weiss and C.T.R. Lai, *Software Product-Line Engineering*. Addison-Wesley, 1999.