Proceedings of the First International Workshop on

Library-Centric Software Design (LCSD '05)

An OOPSLA Workshop

October 16, 2005

San Diego, California, USA

Andrew Lumsdaine and Sibylle Schupp (Program Co-Chairs)
David Musser and Jeremy Siek (Proceedings Editors)

Rensselaer Polytechnic Institute
Computer Science Department
Technical Report 06-12

Foreword

These proceedings contain the papers selected for presentation at the workshop *Library-Centric Software Design* (LCSD'05), held on October 16, 2005 in San Diego, California, USA, as part of the yearly ACM Object Oriented Programming, Systems, Languages and Applications (OOPSLA) conference. This was the first Library-Centric Software Design workshop, and we are pleased that the interest in the workshop was so high.

Software libraries are central to all major scientific, engineering, and business areas, yet the design, implementation, and use of libraries are underdeveloped arts. The goal of the Library-Centric Software Design workshop therefore is to place the various aspects of libraries on a sound technical and scientific basis. To that end, we welcome both research into fundamental issues and the documentation of best practices.

We received 15 papers and were able to select 7 technical papers and 6 position papers. These papers cover a wide range of activities, including theoretical as well as practical questions, along with applications in different languages and paradigms. All papers were reviewed for soundness and relevance by at least three, and in most cases four reviewers. We would like to take this opportunity to thank the program committee for their very thorough reviews, which went far beyond "the usual."

In addition to the paper presentations, the workshop organized a keynote talk, given by Joshua Bloch (Google), and a Birds-of-a-Feather (BOF) session for the discussion of strategic questions. Thirty-two people attended the workshop, and about fifteen the BOF session, from which emerged initial planning for LCSD'06, which will take place Oct. 22, 2006 at OOPSLA in Portland, Oregon.

The idea for a workshop on Library-Centric Software Design was born at the Dagstuhl meeting *Software Libraries: Design and Evaluation* in March 2005. We thank the participants of this meeting for encouraging and nurturing the workshop idea from the beginning; in particular Frank Tip and Bjarne Stroustrup were instrumental in making the LCSD workshop happen. Bjarne initiated, and wrote, the Call for Papers for the workshop.

We would like to thank all authors, reviewers, and the organizing committee for their work in bringing about the LCSD workshop. We are very grateful to David Musser (for serving as the General Chair), Jaakko Järvi (for maintaining the webpage), Dong Inn Kim and the Open Systems Lab at Indiana University (for setting up CyberChair and managing the submissions), and David Musser and Jeremy Siek (for preparing the technical report). We also thank Bill Opdyke and the OOPSLA workshop organizers for the help we received.

We hope you find the papers rewarding and stimulating.

Andrew Lumsdaine
Sibylle Schupp
Program Co-Chairs

Organization

Workshop Organizers

- **Jaakko Järvi**, Texas A&M University
- **Andrew Lumsdaine**, Indiana University
- **David Musser, General Chair**, Rensselaer Polytechnic Institute
- **Sibylle Schupp**, Chalmers University of Technology
- **Jeremy Siek**, Rice University
- **Todd Veldhuizen**, Indiana University

Program Committee

- **Uwe Assman**, Technical University of Dresden
- **Matt Austern**, Google Inc.
- **Hervé Brönnimann**, Polytechnic University
- **Antonio Cisternino**, University of Pisa
- **Jack Dongarra**, University of Tennessee
- **Ulrich Eisenecker**, University of Leipzig
- **Rob Fowler**, HiPerSoft, Rice University
- **Jaakko Järvi**, Texas A&M University
- **Calvin Lin**, University of Texas at Austin
- **Andrew Lumsdaine, co-chair**, Indiana University
- **David Musser**, Rensselaer Polytechnic Institute
- **Sibylle Schupp, co-chair**, Chalmers University of Technology
- **Jeremy Siek**, Rice University
- **Anthony Simons**, University of Sheffield
- **Alex Stepanov**, Adobe Systems Inc.
- **Bjarne Stroustrup**, Texas A&M and AT&T Labs
- **Don Syme**, Microsoft Research
- **Frank Tip**, IBM Research
- **Todd Veldhuizen**, Indiana University

Table of Contents

Technical Papers

What Is Generic Programming? 1
Gabriel Dos Reis, Jaakko Järvi (Texas A&M, USA)

Software Libraries and Their Reuse: Entropy, Kolmogorov Complexity, and Zipf's Law 11
Todd Veldhuizen (Indiana University, USA)

Advanced Programming Techniques Applied to CGAL's Arrangement Package 24
Ron Wein, Efi Fogel, Baruch Zukerman, Dan Halperin (Tel Aviv University, Israel)

Reference Counting in Library Design—Optionally and with Union-Find Optimization 34
Lutz Kettner (MPI, Germany)

A Rationale for Semantically Enhanced Library Languages 44
Bjarne Stroustrup (Texas A&M, USA)

DMTL: A Generic Data Mining Template Library 53
Mohammad Hasan, Vineet Chaoji, Saeed Salem, Mohammed Zaki (Rensselaer Polytechnic Institute, USA)

Changing Iterators with Confidence. A Case Study of Change Impact Analysis Applied to Conceptual Specifications 64
Marcin Zalewski, Sibylle Schupp (Chalmers University, Sweden)

Position Papers

Meta-Driven Library Design 75
Antonio Cisternino (Pisa University), Walter Cazzola (Milano University), Diego Colombo (IMT Lucca, Italy)

Framework Design Using Inner Classes—Can Languages Cope? 80
Kaspar Osterbye (IT University of Copenhagen, Denmark)

The Diary of a Datum: An Approach to Analyzing Runtime Complexity in Framework-Based Applications 85
Nick Mitchell, Gary Sevitsky, Harini Srinivasan (IBM TJ Watson, USA)

A Model for Software Libraries 91
John Hunt, John D. McGregor (Clemson University, USA)

Making a Boost Library 100
Robert Ramey (RRSD.com, USA)

xpressive: Dual-Mode DSEL Library Design 105
Eric Niebler (Boost Consulting, USA)

What is Generic Programming?

Gabriel Dos Reis
Department of Computer Science
Texas A&M University
College Station, TX–77843
gdr@cs.tamu.edu

Jaakko Järvi
Department of Computer Science
Texas A&M University
College Station, TX–77843
jarvi@cs.tamu.edu

Abstract

The last two decades have seen an ever-growing interest in *generic programming*. As for most programming paradigms, there are several definitions of generic programming in use. In the simplest view generic programming is equated to a set of language mechanisms for implementing type-safe polymorphic containers, such as `List<T>` in Java. The notion of generic programming that motivated the design of the Standard Template Library (STL) advocates a broader definition: a programming paradigm for designing and developing reusable and efficient collections of algorithms. The functional programming community uses the term as a synonym for polytypic and type-indexed programming, which involves designing functions that operate on data-types having certain algebraic structures. This paper aims at analyzing core mathematical notions at the foundations of rational approaches to generic programming and library design as reasoned and principled activity. We relate several methodologies used and studied in the imperative and functional programming communities. As a necessary step, we provide a base for common understanding of techniques underpinning generic software components and libraries, and their construction, not limited to a particular linguistic support.

1 Introduction

The notion of "generic programming" has been in use for about four decades, popularized in the '60s with the LISP programming language and its descendents [McC60, ASS84] providing direct support for higher-order functions. Since then, programming techniques and linguistic support for defining algorithms that are capable of operating over a wide range of data structures have been subjects of a large body of work. The notion of *polymorphism* appears to be an essential ingredient of generic programming. In 1967, Christopher Strachey proposed a classification of polymorphism [Str67], based on the linguistic supports present in programming languages. Luca Cardelli and Peter Wegner later refined that classification [CW85], accounting for new language constructs.

Curiously, language features for writing some classes of polymorphic functions and data structures have received more attention than sound programming techniques at the foundation of generic libraries. In fact, generic programming (as usual with successful programming paradigms) is often equated with language features. It is not uncommon to see definitions of "generic programming" that are more or less crafted to mean what the specific programming languages under consideration support [BJJM99]. Similarly, much of the conventions and practice of generic programming in the context of C++ [ISO03, Str00] is shaped by the template system of C++. It is thus difficult to objectively define generic programming without a bias to a particular programming language over others. But if we want to think of generic programming as a principled, reasoned activity, such a language independent understanding is necessary. Consequently, this paper will not focus on language features as the subject of study. The reader interested in a comparison of mainstream programming language features for generic programming is referred to the report of Ronald Garcia *et al.* [GJL+03]. To avoid being lost in the twists and turns of the "empty set theory" we illustrate our ideas and claims with extensive examples written in concrete programming languages, in particular, C++ [ISO03, Str00], Haskell [PJ03], and Scheme [R5R98]. The list of programming languages used in this paper is kept short to avoid distraction. Of course, we hope that the reader would translate or re-express our examples in his or her own favorite programming languages.

Our long term goal is to develop useful theories of generic programming, to better understand and advance the practice of generic programming as a principled activity. This paper reports work in progress along this path, starting from analyzing and relating several notions of generic programming.

It is good to have theories that clarify practice. Good theories, however, are not those that simply rehash common knowledge. Good theories help predict and conquer unexplained and/or unexplored territories. For example, Newton's theory of gravitation was good because it clarified practices and beliefs of the time *but also* helped predict eclipses within reasonable precision. The theories of relativity developed by Einstein were good because they explained facts that left physicists perplexed, and took up where Newton's theory was defeated in predictions. From empirical sciences, one can observe that useful theories are falsifiable. That is, they can be confronted with hard data from the world. Similarly, we posit that useful theories that help gain better understanding of generic programming should be confronted with practices from the real world. The theories are not the goals in themselves, they are means by which we seek to have better understanding. Also, care must be exercised so as not to confuse theories with realities in interpretations.

As its main contribution, this paper shows how different approaches to generic programming can be explained within the same mathematical framework, leaning on category theory. We note that the connection between category theory and generic programming in functional programming languages has been well established — many generic algorithms draw their motivation from categorial notions. A novelty of this paper is the establishment of similar connections for generic programming approach as pioneered by Alexander Stepanov, David Musser and their collaborators (at the foundation of the STL), which arises largely from a practical perspective of organizing generic software components for increased reusability.

The latter approach builds on low level language features — driven by efficiency considerations — much more so than the other approaches to generic programming. As a result, however, proving properties of and reasoning about STL generic algorithms is difficult. We believe a stronger connection to a formal model of generic programming will aid in this respect, guiding the development of generic libraries, and program manipulation tools for them.

2 Background

Generic programming has been approached from various angles in both the functional programming and imperative programming communities. We identify two main schools of thought:

1. the "gradual lifting of concrete algorithms" discipline as first described by David Musser, Alexander Stepanov, Deepak Kapur and collaborators;
2. a calculational approach to programming, the foundations of which were laid by Richard Bird and Lambert Meertens.

The first school defines the discipline of generic programming essentially as follows: start with a practical, useful, algorithm and repeatedly abstract over details; at any stage of the gradual abstraction, the "generic" version of the algorithm shall be such that when instantiated it shall match the original algorithm both in semantics and efficiency. The gradual lifting stops when these conditions cease to hold. Quoting Musser and Stepanov [MS88]:

> By generic programming, we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete efficient algorithms.

The requirement of abstract specification independent of the actual data representation is fundamental for two reasons: 1) it is at the basis of substitution of one datatype interface for another when they are similar; and 2) it allows for classification of similar interfaces based on their efficiency. For example, the linear search function `find()` of the Standard Template Library [SL94] works on iterators coming from either a linked-list or an input stream because they provide similar interfaces for increment and value-fetching. However, `binary_search()` is defined only for forward iterator interfaces.

The second school of thought in generic programming has its root in the initial algebra approach to datatypes as advocated by Joseph Goguen and collaborators [GTWW77, TWW82] and a calculational approach to program construction [Bir87, Mee86]. Category theory is an essential tool in this setting. In "Generic Programming — an Introduction" [BJJM99], Roland Backhouse *et al.* stated:

> we introduce another dimension to the level of abstraction in programming languages, namely parameterization with respect to classes of algebras of variable signature.

In this approach, also referred to as *datatype generic programming*, structures of datatypes are parameters of generic programs. Datatype generic programming [JJ96, JJ97, BJJM99, Hin00, Hin04] has had a strong focus on regular datatypes essentially described by algebras generated by the functors *sum*, *product* and *unit*. Algorithms written for those functors can then operate on any inductive datatype, and are thus inherently very generic. Indeed, a fairly large class of generic algorithms can be defined in this manner, such as structural equality, serialization/deserialization, zips, folds, and traversals.

The Musser–Stepanov style of generic programming emphasizes *concept analysis*, the process of finding and establishing the important classes of concepts that enable many useful algorithms to work. Programmers then explicitly define correspondence from their datatypes to those classes of concepts. A thesis of this paper is that concept analysis is a way of looking for functors that capture common structures. We can see that the two definitions of generic programming are fundamentally very close to each other, but the emphasis in each view is on different aspects: one focusing on a particular structural algebra for datatypes and the algorithms defined in terms of that algebra, whereas the other on finding and classifying classes of algebras based on some notions of efficiency.

Finally, we can observe that while both methodologies have an underlying theoretical language-independent model, C++ has become the dominating platform for the Musser–Stepanov style[1], whereas Haskell and its variants are the almost exclusive tool for data-type generic programming.

3 Using category theory

Category theory is a branch of mathematics originally developed as a language to unify and abstract over many structure and proof patterns in Algebraic Topology. Category theory — also occasionally referred to as "abstract nonsense" or "the theory of empty set" — has found an unreasonably effective application in Computer Science. The theoretical core ideas of the categorial approach to datatypes and generic functions go back at least to Goguen and collaborators [GTWW77].

3.1 Elementary notions

This section recalls some basic notions of category theory and establishes vocabulary used in the rest of the paper. We have kept the load of jargon to the minimum; the reader interested in further development of category theory might advantageously consult the standard textbook of Saunders Mac Lane [ML01]. Within the discussion, we include examples of how the categorial notions become manifest as idioms and patterns in practical programming.

3.1.1 Categories

A *category* $\mathcal{C}$ is a collection of *objects* and *arrows* (also called *morphisms*) between objects with three fundamental operations:

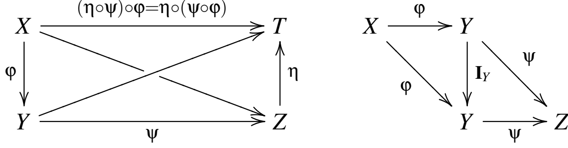
1. Every arrow φ in $\mathcal{C}$ is associated with two objects:
   - its *source* $\operatorname{dom}\varphi$, an object of $\mathcal{C}$, and
   - its *target* $\operatorname{cod}\varphi$, also an object of $\mathcal{C}$.

   Thus, an arrow is often written as $\varphi : X \to Y$, where X is the source and Y the target.
2. Every object X in $\mathcal{C}$ is associated with a distinguished arrow $\mathbf{I}_X : X \to X$, called the identity arrow of object X.

[1] Though Musser's and Stepanov's early work on generic programming was in the context of Scheme and Ada.

3. For two composable arrows $\varphi : X \to Y$ and $\psi : Y \to Z$ in $\mathcal{C}$, the composition $\eta = \psi \circ \varphi : X \to Z$ is again an arrow in $\mathcal{C}$.

   Furthermore, the composition operator must be associative and admits the identity arrow as unit, which diagrammatically reads



The collection of arrows from an object X to an object Y is called the *hom-set* from X to Y and written $\hom_{\mathcal{C}}(X,Y)$. The subscript is used to emphasize the category under consideration.

3.1.1.1 Examples

Small sets Our first example of a category is **Set** whose objects are sets and arrows are the usual total functions between sets.

Complete partial orders Recall that a *partial order* $\preccurlyeq$ on a set X is a binary relation on X that is reflexive, transitive and anti-symmetric. A set equipped with a partial order is said a *partially ordered set* or *poset* for short. For example, the set $\mathbb{N}$ of natural numbers equipped with the relation "divides" is a poset. A function f from a poset $(X, \preccurlyeq_X)$ to a poset $(Y, \preccurlyeq_Y)$ is said *monotonic* if $f(x_1) \preccurlyeq_Y f(x_2)$ whenever $x_1 \preccurlyeq_X x_2$. An ω-*chain* in a poset X is a sequence $x : \mathbb{N} \to X$ such that $x_i \preccurlyeq x_{i+1}$. A poset in which every ω-chain has a least upper bound is called an ω-*complete poset*. An ω-complete poset with a least element is said to be an ω-*complete pointed poset*. For example, the power set 2^A of a set A is an ω-complete pointed poset when equipped with inclusion as partial order.

A *continuous* function between two posets is a monotonic function that sends the least upper bound of an ω-chain to the least upper bound of the image of the chain. The collection **CPO** of ω-complete pointed posets is a category where the arrows are continuous functions; $\mathbf{CPO}_\perp$ is a **CPO** with a least element.

3.1.2 Initial and terminal objects

An object $\mathfrak{i}$ is called *initial* in a category $\mathcal{C}$ if, for every object X in $\mathcal{C}$, the hom-set $\hom_{\mathcal{C}}(\mathfrak{i},X)$ is a singleton. Dually, an object $\mathfrak{t}$ is said to be *terminal* if for every object X in $\mathcal{C}$, the hom-set $\hom_{\mathcal{C}}(X,\mathfrak{t})$ is a singleton. A category can admit at most one initial (resp. terminal) object, up to isomorphism.

3.1.2.1 Examples

In **Set**, the empty set **0** is initial. On the other hand, every singleton **1** is terminal.

3.1.3 Functors

Categories are not very interesting by themselves; what is interesting about them is *what* is happening in or between them, *e.g.* functors, etc. that we will define shortly. When studying structures, the first natural thing one usually does is to look for properties that remain unchanged over similar structures. For categories, that means properties that remain unchanged through the composition operator in a class of structures.

A *functor* F from a category $\mathcal{C}$ to a category $\mathcal{D}$ is a morphism of categories; it consists of two parts:

1. An *object function* which assigns an object $F(X)$ in $\mathcal{D}$ to every object X in $\mathcal{C}$;
2. An *arrow function* that assigns an arrow $F(\varphi) : F(X) \to F(Y)$ in $\mathcal{D}$ to every arrow $\varphi : X \to Y$ in $\mathcal{C}$ such that
   - the identity arrow is sent to the identity arrow, *i.e.*,
     $$F(\mathbf{I}_X) = \mathbf{I}_{F(X)}$$
     for every object X in $\mathcal{C}$,
   - two composable arrows $\varphi : X \to Y$ and $\psi : Y \to Z$ are sent to composable arrows and the property
     $$F(\psi \circ \varphi) = F(\psi) \circ F(\varphi)$$
     holds.

We will say that $F(\varphi)$ is the *lift* of the arrow φ by F.

3.1.3.1 Examples

Identity functor A ubiquitous functor is the identity functor **I**. Both its object function and arrow function yield their arguments unchanged.

Constant functor Any object A in a category $\mathcal{C}$ gives rise to a functor $\mathbb{A}$ as follows: the object function sends all objects to A, and the arrow function sends all arrows to the identity arrow of A. In particular, "the" singleton object **1** gives rise to the unit functor $\mathbb{1}$.

3.1.4 Multivariate functors

The notion of functor can be generalized to that of *bifunctor*, operating simultaneously on two categories so that the composition law holds component-wise:

$$F(\varphi_2 \circ \varphi_1, \psi_2 \circ \psi_1) = F(\varphi_2, \psi_2) \circ F(\varphi_1, \psi_1).$$

3.1.4.1 Examples

For the purpose of this paper, we will assume that we are mostly working in $\mathbf{CPO}_\perp$. This simplifies the exposition allowing us to talk about least and greatest fixed points, making the connection to algebras and co-algebras less heavy-weight. The functor examples given in this section could, however, be defined in a more general setting by universal property, i.e., by singling out specific objects with unique arrows to or from them.

Product functor A commonly used functor is the product functor. Its object function sends two objects X and Y to the object

$$X \times Y = \{(x,y) \mid x \in X, y \in Y\}$$

and its arrow function sends two arrows $\varphi : X \to S$ and $\psi : Y \to T$ to the arrow $\varphi \times \psi : X \times Y \to S \times T$ defined by

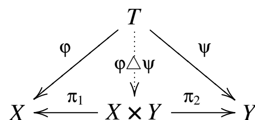
$$(\varphi \times \psi)(x,y) = (\varphi(x), \psi(y)).$$

It can be readily verified that $\times$ indeed is a bifunctor.

The product functor is concretely realized in programming languages in various ways. In C++ for instance, the object function is implemented by the standard library class template `std::pair<X, Y>`. However, there is no predefined arrow function. One can be literally defined as

```
template<class X, class Y, class S, class T>
std::pair<S, T>
lift(const std::pair<X, Y>& p, S f(X), T g(Y))
{
  return std::pair<S, T>(f(p.first), g(p.second));
}
```

Associated with the product functor are the projection combinators π_1 and π_2 leading to the tupling combinator $\triangle$ that makes the following diagram commute



for any pair of arrows $\varphi : T \to X$ and $\psi : T \to Y$.

In code, the tupling combinator would read

```
template<class T, class X, class Y>
std::pair<X, Y> tuple(T t, X f(T), Y g(T))
{
   return std::pair<X, Y>(f(t), g(t));
}
```

Sum functor Yet another commonly used functor is the discriminated union. It takes objects to *tagged pairs*

$$X+Y = \{0\} \times X \cup \{1\} \times Y \cup \{\bot\}$$

and arrows to arrows *defined by case analysis*

$$(\varphi+\psi)(\bot) = \bot$$
$$(\varphi+\psi)((0,x)) = (0,\varphi(x))$$
$$(\varphi+\psi)((1,y)) = (1,\psi(y))$$

where a pattern matching is done as follows: if the argument is junk, then it is returned untouched; if the argument was built from an element of the first component then it is extracted, given to the first arrow and the result is packaged back into the first component; otherwise if the argument was built from an element of the second component then it is extracted, given to the second arrow and the result is packaged back into the second component.

The above behavior takes lots of words to describe but very few symbols to define in Haskell

```
data Either a b = Left a | Right b
eitherLift ::
  (a -> c) -> (b -> d) -> Either a b -> Either c d
eitherLift f g (Left x)  = Left (f x)
eitherLift f g (Right y) = Right (g y)
```

Discriminated unions are idiomatically expressed in languages without built-in pattern matching as instances of the Visitor Design Pattern [GHJV94]. In C++ for example, using this scheme we define a base class `Either` with derived classes `Left` and `Right`. A class `EitherVisitor` that can visit classes derived from `Either` is also needed.

```
template<class X, class Y> class Either;
  template<class X, class Y> class Left;
  template<class X, class Y> class Right;

template<class X, class Y>
struct EitherVisitor {
  virtual void visit(const Left<X, Y>&) = 0;
  virtual void visit(const Right<X, Y>&) = 0;
};

template<class X, class Y>
struct Either {
  virtual ~Either() { }
  virtual void accept(EitherVisitor<X, Y>& v) const = 0;
};

template<class X, class Y>
struct Left : Either<X, Y> {
  const X& x;
  Left(const X& x) : x(x) { }
  void accept(EitherVisitor<X, Y>& v) const
    { v.visit(*this); }
};

template<class X, class Y>
struct Right : Either<X, Y> {
  const Y& y;
  Right(const Y& y) : y(y) { }
  void accept(EitherVisitor<X, Y>& v) const
    { v.visit(*this); }
};
```

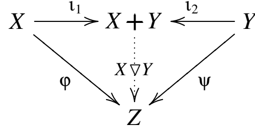
The code has a fair amount of boilerplate to simulate pattern matching. Now, the lift mapping itself can be defined as

```
template<class X, class Y, class S, class T>
const Either<S, T>
lift(const Either<X, Y>& e, S f(X), T g(Y))
{
   typedef S (*F)(X);
   typedef T (*G)(Y);
   struct Impl : EitherVisitor<X, Y> {
      F f;
      G g;
      const Either<S, T>* value;
      Impl(F f, G g) : f(f) g(g), value() { }

      void visit(const Left<X, Y>& e)
      {
         value = left<S, T>(f(e.x));
      }
      void visit(const Right<X, Y>& e)
      {
         value = right<S, T>(g(e.y));
      }
   };

   Impl vis(f, g);
   e.accept(vis);
   return *vis.value;
}
```

We use helper functions `left<S, T>()` and `right<S, T>()` for allocating objects of the obvious types. The code is undoubtly more involved than the corresponding few lines in Haskell (or ML). It is

not intended as a translation of Haskell to C++, but as illustration of both basic categorial constructs and common techniques used in languages lacking direct support for pattern matching.

Dually to the case of product, the sum functor comes with two injection combinators ι_1 and ι_2 and a conflating combinator ∇ making

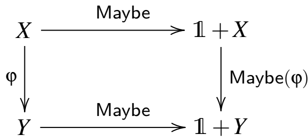


a commutative diagram, for any pair of arrows $\varphi : X \to T$ and $\psi : Y \to T$.

In code, the destruction combinator is typically given by case analysis (because its domain is a discriminated union).

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  = f x
either f g (Right y) = g y
```

The Maybe **functor** It is the functor $\mathbb{1} + \mathbf{I}$ whose action is described diagrammatically as



Conceptually, it describes the type of objects that may hold values of another datatype or nothing.

3.1.5 *Algebras and co-algebras*

In this section we consider only endofunctors, *i.e.*, functors with identical sources and targets.

3.1.5.1 *Algebras*

The notion of algebra generalizes that of Σ-algebra from the theory of Universal Algebra [Coh81] where an algebra can be thought of as interpretation of a collection of function symbols, and the structures of their domains are given by the functor.

Given an endofunctor F of a category $\mathcal{C}$, an arrow of the form

$$\alpha : F(X) \to X$$

is called an *F-algebra* — written $(\alpha, X)_F$ or simply (α, X) when the functor is understood from context — and the object X is its *carrier*.

In $\mathbf{CPO}_\perp$ for example, if one thinks of a polynomial functor as describing a structure X together with operation symbols, then an algebra appears as an interpretation by case analysis.

Example The Haskell datatype

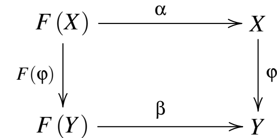
```
data Nat = Zero | Succ Nat
```

is a Maybe-algebra, because the above definition introduces the operation `Zero` $\curlyvee$ `Succ` where

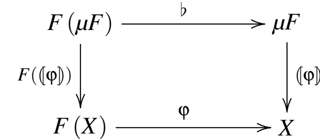
```
Zero :: Nat        -- can be thought as Zero :: 𝟙 -> Nat
Succ :: Nat -> Nat -- successor operation
```

Here we would like to interpret `Zero` as the natural number 0, and `Succ` as the operation that yields the successor of a natural number. Of course, that is not the only possible interpretation; but among all possible interpretations, there is a distinguished one. We make that idea more precise in the following paragraphs.

Given an endofunctor F on a category $\mathcal{C}$ and two F-algebras (X, α) and (Y, β), an arrow $\varphi : X \to Y$ that makes



a commutative diagram, *i.e.*, $\varphi \circ \alpha = \beta \circ F(\varphi)$, is called an *F-algebra homomorphism*. The collection $\mathbf{Alg}(F)$ of F-algebras can be readily seen to form a category where the arrows are the F-algebra morphisms. The initial object $(\mu F, \flat)$ of that category, when it exists, is called the *initial F-algebra*. It has the distinguishing characteristic that given any F-algebra (φ, X) there is unique F-algebra homomorphism — written $(\!|\varphi|\!)$ — from μF to X making the diagram



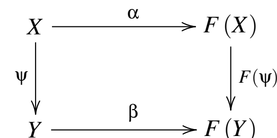
commutative. The arrow $(\!|\varphi|\!)$ is said to be the *catamorphism* of φ. Examples of catamorphisms will be given in §3.2.1

3.1.6 *Coalgebras*

A *coalgebra* is the dual notion of an algebra, *i.e.*, an arrow of the form

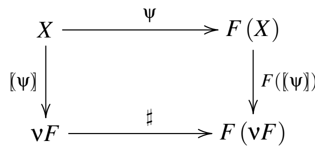
$$\alpha : X \to F(X)$$

which we will denote by $[\alpha, X]_F$. One can also define the notion of *F-coalgebra homomorphism* which is an arrow $\psi : X \to Y$ that makes the diagram



commute for any pair of F-coalgebras α and β. The collection $\mathbf{CoAlg}(F)$ of F-coalgebras, with F-coalgebra homomorphisms as arrows, is a category. The terminal object $(\nu F, \sharp)$ of that category, when it exists, is called the *final coalgebra* of the functor F. It is characterized by the fact that given any F-coalgebra $[\psi, X]$, there corresponds a unique F-coalgebra homomorphism from X to νF

that makes



a commutative diagram. The F-coalgebra $[\![\psi]\!]$ is called the *anamorphism* of the arrow ψ.

3.2 Categorial datatypes

3.2.1 Initial datatypes

The initial algebraic approach to datatypes posits that when working in an appropriate category, many abstract data types are nothing but initial algebras of some functor. For example, the usual set of natural numbers as described by the Peano axioms is the initial algebra of the functor Maybe.

The main benefit of viewing datatypes as initial algebras is that an iteration operator over the datatypes, called *fold*, follows for free. That crucial property provides a convenient *implementation* tool and *reasoning* device to capture patterns. In $\mathbf{CPO}_\perp$ for instance, it can be shown that every polynomial functor has an initial algebra, which in fact is its least fixed point.

For example, consider the bifunctor

$$S(T,X) = \mathbb{1} + T \times X = \mathsf{Maybe}(\mathsf{T} \times \mathsf{X}).$$

Its least fixed point with respect to the second argument yields an object parameterized by T

$$Seq(T) = \mathbb{1} + T \times Seq(T)$$

which captures many algebraic aspects of *finite sequences* of values of type T. When viewed as acting on T, it can be thought of as a functor; we will call it the *sequence functor*. A *cons-list* from functional programming practice is an example of such an object. In Haskell, it is defined by

```
data List a = Nil | Cons a (List a)
```

For a fixed T, $Seq(T)$ is the least fixed point of the functor $X \mapsto \mathbb{1} + T \times X$. Computing the length of such list is readily implemented by

```
length :: List a -> Int
length Nil         = 0
length (Cons a as) = 1 + length as
```

where it is apparent that the `length` function is obtained by sending the unit value ($\mathbb{1}$) to 0 and the list constructor `Cons` to the successor operation. That is the essence of catamorphisms, *i.e.*, mapping constructors to functions. Note how that description is an abstract specification of the following C++ algorithm:

```
template<class Forward>
int length(Forward first, Forward last)
{
  int n = 0;
  for (; first != last; ++first)
     ++n;
  return n;
}
```

The fundamental operations of the functor *Seq* are materialized here by

- when to stop or empty sequence $\mathbb{1} \leftrightarrow$ `first == last`;
- next elements of the sequence `++first`.

Then the mapping corresponds to initialization to 0 and incrementation respectively. The act of replacing a signature (here 0 and the successor functions) with a function is the essence of catamorphism, and the basis of polytypic functions. The STL algorithm `accumulate` is the *fold* for sequences, and many other STL algorithms are specializations of it.

3.2.2 Final datatypes

Final datatypes are dual to initial datatypes. They can be modeled as final coalgebras. In the category $\mathbf{CPO}_\perp$, the final coalgebra of a polynomial functor is its *greatest* fixed point. For example, the greatest fixed point of the functor

$$X \mapsto T \times X$$

is the *infinite list* or *stream* of values of types T, characterized by two fundamental operations

$$\mathtt{head} : Stream(T) \to T$$
$$\mathtt{tail} : Stream(T) \to Stream(T).$$

The C++ standard iterator `ostream_iterator<>` is a genuine example of handles to streams — there is no way to test for "stopping conditions".

The greatest fixed point of the $X \mapsto \mathsf{Maybe}(\mathsf{T} \times \mathsf{X})$ (see §3.2.1) is a *potentially infinite list*. Unlike the case for streams, one can test a potentially infinite list for stopping conditions.

The main difference between initial datatypes and final datatypes is that the former are characterized by constructors whereas the latter are characterized by observers and modifiers.

4 Recursion patterns

The categorial approach to data types makes clear connections between the patterns of "regular" recursive algorithms and those of data types. The most popular being catamorphism, anamorphism and hylomorphism (an anamorphism followed by a catamorphism) [MFP91]. Interestingly, such patterns are essentially present in the Musser–Stepanov approach to structure algorithms, in slightly different forms (iterative mostly) and spelled out differently. Consider the following function template `accumulate` from the STL:

```
template <class Input, class T, class BinOp>
T accumulate(Input first, Input last, T init, BinOp op)
{
  for (; first != last; ++first)
     init = op(init, *first);
  return init;
}
```

This function essentially defines what corresponds to a fold, the general recursion operator for defining catamorphisms, over a `List` functor. Compare this to the typical definition of a fold in, say, Haskell:

```
foldr           :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

When `foldr` is called with a mapping for the data constructors list, we get specific catamorphisms. This is directly visible in the Haskell case: the mapping `z` is applied to lists constructed with `[]`, and `f` to lists constructed with the cons operator `:`. We use `foldr` (instead of `foldl`) because it is the natural iteration operation for the list datatype as defined in Haskell. For example, `foldr (+) 0 a:(b:(c:[]))` gives, after mapping `+` and `0` appropriately, `a+(b+(c+0))`.

In the C++ version, `init` corresponds to `z`, the empty list is denoted by the negation of `first != last`, and `op` is the same as `f`. As an example, in Haskell, the catamorphism `length` for computing the length of a list is obtained by mapping `0` to the empty list, and an increment function to the cons constructor:

```
length ls = foldr (1+) 0 ls
```

Analogously, the C++ `length` function can be written in terms of `accumulate` as follows:

```
struct incrementor {
  template<class X, class Y>
  X operator()(X x, const Y& t) const { return x + 1; }
};

template <class In>
int length(In first, In last)
{
  return accumulate(first, last, 0, incrementor());
}
```

With the help of a library that provides convenient notation [JPL03], one can simply write

```
template <class In>
int length(In first, In last)
{
  return accumulate(first, last, 0, _1 + 1);
}
```

Many other STL algorithms — `for_each`, `transform`, and `find` to name a few — can be defined as catamorphisms using `accumulate`. The view of a fold as a combinator that defines a traversal, or recursion pattern, for algebras with a particular signature, applies equally well in the context of STL, as it does in the context of Bird–Meertens formalism. However, whereas generalized folds over all regular data types, such as binary trees, are possible in data-type generic programming, this is not the case for STL. For example, `accumulate` is defined only for sequences, not for algebras describing binary trees. As a remedy, STL defines a homomorphism from binary trees (the `map` data structure implemented as red-black trees) to sequences, but this does not enable generic algorithms that truly operate on the structure of the tree. In particular, the homomorphism fixes in-order as the only traversal for STL maps. There are practical consequences of this. For example, copying a STL map to another map with the `std::copy` algorithm exhibits worst case complexity in terms of necessary rotations in the underlying red-black tree. Similarly, the generic `find` algorithm cannot take advantage of the special structure of the tree.

5 Transforming sequences

In line with our "meta" views developed in the opening of this report, we start with the simple idea of transforming a sequence into another one by applying a given function to each element. For concreteness, here is a Scheme routine for that:

```
(define (map function sequence)
  (cond ((null? sequence) nil)
        (else (cons (function (car sequence))
                    (map function sequence)))))
```

That definition assumes the ubiquitous, built-in, Scheme datatype of list to represent a sequence of items. The program fragment inspects its input with the observers

- `null?` to test for an empty sequence;
- `car` to inspect the value of the head of a sequence;
- `cdr` to get to the remaining items in a sequence;

and constructs its output with:

- `cons` to construct a new sequence out of an existing item and a sequence.

These operations seem to be fundamental primitives needed to write the algorithm as a Scheme program. Data constructors (*e.g.* `cons`) are typical to initial algebra treatment of generic datatypes and functions, whereas observers (*e.g.* `null?`, `car`, `cdr`) are defining characteristics of final coalgebras. Consequently, this expression of the transformation function makes a mixture of initial algebras and final coalgebras. Is that mixture essential to capture the algebraic essence of `map`? We will see a purely initial algebraic formulation in §5.1. If not, is that mixture essential to make `map` operate on a wider class of sequence implementations? A fundamentally final coalgebraic definition is given in §5.2 as a C++ function that operates on a wide variety of sequence instances.

The definition of `map` has a direct imprint of the built-in list type — uses of `null?`, `car`, `cdr` and `cons` that have built-in meaning. As is, it is not usable with another incarnation of sequences, say with `vectors`. However, that limitation can be overcome in several ways. One way is to use symbols — *e.g.* `empty?`, `head`, `tail`, `new-seq` and `null` — that can support the abstract operations on a variety of sequence implementations, based on the "data-directed" programming paradigm [ASS84]. In that perspective, their implementations would abstract away the differences in sequence implementations through runtime type-based dispatch. In C++ such an approach could be expressed through overloading or overriding, whereas in Haskell it would take the form of type classes.

Another way of removing the limitation is via higher-order functions, passing the necessary operators as parameters:

```
(define (map fun seq empty? head tail new-seq null)
  (cond ((empty? seq) null)
        (else (new-seq (fun (head seq))
                       (map fun (tail seq) empty?
                            head tail new-seq null)))))
```

This version is fully general and makes no hard-coded assumptions on how the sequence is represented. However, the function may be awkward to use. In particular, every use site of this function must ensure that the right operations are passed along with the right

sequence implementations. For example, calling `map` with a list and `vector-ref` will lead to (runtime) error. We see that what we need here is a way of referring to the iteration operator of the *concrete implementation* of the notion of sequence.

This new version of `map`, as well as the first, features several issues in generic programming — accessors as final coalgebras and constructors as initial algebras.

5.1 A slightly different look at map

The `map` function is also part of the Haskell Prelude [PJ03] and defined as

```
map :: (a -> b) -> [a] -> [b]
map f []        = []
map f (x : xs)  = f x : map f xs
```

The explicit type annotation makes it unambiguous that `map` is defined to work only on Haskell's built-in datatype list. The only genericity achieved here is the variability of the contained element type. However, as we observed in the previous section, the notion of transformation is not restricted to a specific instance of the notion of sequence.

This expression of `map` uses a slightly different approach. Namely, it accesses the building blocks of the input sequence through pattern matching. Therefore it makes essential use of the list data constructors, and is completely defined in terms of those. It can be completely characterized in terms of the initial algebra for list (which really has a stack implementation in most functional programming languages). Consequently, while the definition works on the list implementation of the notion of sequence, it does not work on the Haskell `Array` implementation or any other sequence implementation that does not use the list constructors.

To overcome the use of built-in constructors that tie `map` to a given data type, the Haskell library uses a type class `Functor` as implementation of the general notion of functor, as discussed in §3.1.3:

```
class Functor f where
    fmap :: (a -> b) f a -> f b
```

The idea is that the symbol `fmap` will be applicable to all type constructors for which there are known instance declarations stating that they act like functors. Given such a declaration, a use of `fmap` on a particular concrete sequence is to be made in conjunction with `Functor` instance declarations for that concrete sequence implementation. This situation reminds us of the drawbacks typical of object oriented programming where operations are closely tied to objects (*e.g.* member functions) so that writing N algorithms for P datatypes requires solving $N \times P$ problems.

Applying Stepanov–Musser's methodology to lift `map` to more generic levels, capable of operating over a wider range of data types, requires giving up specific knowledge of the built-in list type. As a consequence, the expression of the idea of sequence transformation seems to become more involved. To what extent are the added complexities intrinsic to `map` as opposed to language artifacts? Is the increase of complexity a sign of useful generality gain?

5.2 Yet another look at map

In this section, we look at the expression of the `map` that in the C++ community is known as the standard algorithm `transform`:

```
template<class In, class Out, class Oper>
Out transform(In first, In last, Out out, Oper op)
{
  for (; first != last; ++first)
     *out++ = op(*first);
  return out;
}
```

It is standard, in programming with C++, to represent sequences as pairs of iterators; thus generic sequence algorithms operate on such representations, as laid out in the STL [SL94]. The operations of reading the head of a sequence and moving to the remaining parts are implemented by `*` and `++` operators. The C++ version of `transform` does not use list (sequence) constructors to build the result. Rather, the formulation uses accessors, as if the view is that of *final datatypes*. Consequently, the algorithm can work on all instances of iterators (therefore sequence instances) that provide similar interfaces. The complexity in terms of the number of concrete sequence implementations and concrete transformation implementations is significantly reduced.

6 Limitations

The semi-open interval model used in the STL to represent sequences leaves some data structures out of the picture, most notably circular lists. Similarly, circular list appears to resist initial data type formulations. In fact, circular lists appear to be more amenable to formalization through final coalgebras [Kam83].

What do we gain from the category theory approach to generic programming? Is it effective? What does it explain and what does it predict?

In our view, the categorial approach seeks to capture common algebraic structures, similarities of interfaces as advocated by Dehnert and Stepanov [DS98] (see Section 7). For example, the *fold()* iteration operators are *implementation* tools and *reasoning* devices for capturing traversal and proof patterns common to a class of generic functions [Hut98]. We find the categorial approach as a promising starting point for a theory that can clarify and explain the practice of Musser–Stepanov style generic programming. Moreover, we believe, in accordance with what we state in the introduction of this paper, that the mathematical framework is sufficient for prediction and conquering new grounds as well such as STL in parallel and distributed programming contexts. Along those lines, we mention that libraries and compiler frameworks [RG03] based on the calculational approach, from functional programming perspective, are subjects of active research.

7 Discussion

In this section, we examine, within the mathematical framework in place, the main two approaches to generic programming. The purpose is to identify commonalities and differences in more definite terms.

Dehnert and Stepanov [DS98] advocate maximizing reuse of software components through alikeness identification:

> [...] Breadth of use, however, must come from the separation of underlying data types, data structures, and algorithms, allowing users to combine components of each sort from either the library or their own code. Accomplishing this requires more than just simple, abstract interfaces — it requires that a wide variety of components share the same interface so that they can be substituted for one another. It is vital that we go beyond the old library model of reusing identical interfaces with pre-determined types, to one which identifies the minimal requirements on interfaces and allows reuse by similar interfaces which meet those requirements but may differ quite widely otherwise. Sharing similar interfaces across a wide variety of components requires careful identification and abstraction of the patterns of use in many programs, as well as development of techniques for effectively mapping one interface to another.

Separating data structures from algorithms is key to reducing the complexity of implementing N algorithms for P data structures, as exemplified by the STL. At first sight, that seems to run contrary to the practice of the calculational approach which puts emphasis on iteration operators (folds) intimately associated with recursive data structures. However, it should be observed that once the class of algorithms of interest is identified (*e.g.* sequence algorithms) the iteration operator is also fixed. Other data structures "just" need to have their iteration operators adapted or mapped to the iteration scheme of reference. For example, in the STL all sequences as well as *associative containers* (binary trees in disguise) provide means to iterate *linearly* over them.

The "minimal requirements" tip translates to "final coalgebras" in our framework. That aspect is unlike the approach of the Bird–Meertens formalism, which has been traditionally based on "initial algebras."

Dehnert and Stepanov [DS98] continue:

> We call the set of axioms satisfied by a data type and a set of operations on it a *concept*. Examples of concepts might be an integer data type with an addition operation satisfying the usual axioms; or a list of data objects with a first element, an iterator for traversing the list, and a test for identifying the end of the list. The critical insight which produced generic programming is that highly reusable components must be programmed assuming a minimal collection of such concepts, and that the concepts used must match as wide a variety of concrete program structures as possible. Thus, successful production of a generic component is not simply a matter of identifying the minimal requirements of an arbitrary type or algorithm — it requires identifying the common requirements of a broad collection of similar components. The final requirement is that we accomplish this without sacrificing performance relative to programming with concrete structures.

We can contrast the above to a characterization of abstract data types as classes of algebras. According to Thatcher *et al* [TWW82]:

> what is "abstract" about an abstract data type is that it consists of an isomorphism class of algebras rather than any concrete representation of the class. When it comes to specifying an abstract data type one can display a particular algebra and define the abstract data type as the isomorphism class of that algebra. The proposed alternative is to characterize the isomorphism class using axioms written in terms of the operations on the types.

A fundamental difference between the first school and the second school is that the latter equates linguistic support with generic programming, while the former defines it as a methodology. Furthermore, the Musser–Stepanov school promotes structuring components based on the efficiency or algorithmic complexity offered by the coalgebras, whereas those concerns appear to be secondary in the calculational approach. For example, the data structure list is usually taken as *the* canonical realization of sequences in the functional programming setting. We are not aware of work in the calculational approach, where complexity guarantees of operations (in the style of Musser–Stepanov) and genericity are given equal weight.

In a sense, the opposition of styles is similar to that of bottom-up versus top-down design. From our perspective, a good theoretical framework for generic programming should provide for mathematical tools necessary for systematic application of Dehnert and Stepanov's methodology to both the implementation and correctness proof of generic components as exhibited by the Bird–Meertens formalism.

8 Conclusions

The two approaches to generic programming, 1) as defined by the process and outcome of designing STL and similar libraries, and 2) as defined by the practice of data-type generic programming in the functional programming community, are intrinsically connected. The first approach to generic programming focuses on finding useful fundamental algebras, and defining generic functions mapping to such algebras following a final coalgebra point of view. The second, datatype generic programming, operates on initial algebras and focuses on finding algorithms on those algebras. These algorithms are applicable to a wide variety of data-types, as there are conversions from regular inductive datatypes to the structures of the functors that define them. The most interesting aspect of datatype generic programming is iteration operators for free as implementation tools and reasoning devices to capture patterns in proofs about generic functions. Combining Musser–Stepanov's methodology with a categorial approach to datatypes appears to be a promising road for systematic implementation and proof of properties about useful generic programming, and is subject for future work.

9 Acknowledgments

We are grateful to the anonymous reviewers for their comments and suggestions that improved the paper. We are grateful to Bjarne Stroustrup for suggesting the *doggiemorphism* recursion pattern, which regretfully did not fit within the space limits.

10 References

[ASS84] Hal Abelson, Jerry Sussman, and Julie Sussman. *Structure and interpretation of Computer Programs*. MIT Press, 1984.

[Bir87] Richard Bird. *Logic of Programming and Calculi of Discrete Design*, volume F.36 of *NATO AI Series*, chapter An introduction to the theory of list. Springer Verlag, 1987.

[BJJM99] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, chapter Generic Programming — An introduction, pages 28–115. Springer-Verlag, 1999.

[Coh81] Paul Cohn. *Universal Algebra*. Kluver, 1981.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[DS98] James C. Dehnert and Alexander Stepanov. Fundamentals of Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11, Schloss Dagstuhl, Germany, April 1998.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[GJL+03] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A Comparative Study of Language Support for Generic Programming. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 115–134. ACM Press, 2003.

[GTWW77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial Algebra Semantics and Continuous Algebra. *Journal of the Association of Computing Machinery*, 24(1):68–95, January 1977.

[Hin00] Ralf Hinze. A New Approach to Generic Functional Programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 119–132, Boston, USA, 2000. ACM Press.

[Hin04] Ralf Hinze. Generics for the masses. In *Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming*, pages 236–243, Snow Bird, UT, USA, 2004.

[Hut98] Graham Hutton. Fold and Unfold for Program Semantics. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 280–288, Baltimore, Maryland, USA, 1998.

[ISO03] International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.

[JJ96] Johan Jeuring and Patrik Jansson. Polytypic Programming. In *Advanced Functional Programming, Second International School — Tutorial Text*, volume 1129 of *Lecture Notes In Computer Science*, pages 68–114. Springer-Verlag, 1996.

[JJ97] Patrik Jansson and Johan Jeuring. Polyp — a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 1997.

[JPL03] J. Järvi, G. Powell, and A. Lumsdaine. The Lambda Library: unnamed functions in C++. *Software—Practice and Experience*, 33:259–291, 2003.

[Kam83] Samuel Kamin. Final Data Types and Their Specification. *ACM Transaction on Programming Languages*, 5(1):97–123, January 1983.

[McC60] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of The ACM*, April 1960.

[Mee86] Lambert Meertens. Algorithmics — toward programming as a mathematical activity. In J.W. de Bakker and J.C. van Vliet, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334, North-Holland, 1986.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[ML01] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 2nd edition, 2001.

[MS88] David A. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Rome, Italy, 1988.

[PJ03] Simon Peyton Jones. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[R5R98] Revised[5] Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[RG03] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[SL94] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report N0482=94-0095, ISO/IEC SC22/JTC1/WG21, May 1994.

[Str67] Christopher Strachey. Fundamental Concepts in Programming Languages. Lecture notes for the International Summer School in Computer Programming, August 1967.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[TWW82] J.W. Thatcher, E.G. Wagner, and J.B. Wright. Data Type Specification: Parameterization and the Power of Specification Techiniques. *ACM Transaction on Programming Languages and Systems (TOPLAS)*, 4(4):711–732, October 1982.

Software Libraries and Their Reuse: Entropy, Kolmogorov Complexity, and Zipf's Law

[Extended Abstract]

Todd L. Veldhuizen
Open Systems Laboratory
Indiana University Bloomington
Bloomington, IN, USA
email: tveldhui@acm.org

Abstract

We analyze software reuse from the perspective of information theory and Kolmogorov complexity, assessing our ability to "compress" programs by expressing them in terms of software components reused from libraries. A common theme in the software reuse literature is that if we can only get the right environment in place— the right tools, the right generalizations, economic incentives, a "culture of reuse" — then reuse of software will soar, with consequent improvements in productivity and software quality. The analysis developed in this paper paints a different picture: the extent to which software reuse can occur is an intrinsic property of a problem domain, and better tools and culture can have only marginal impact on reuse rates if the domain is inherently resistant to reuse. We define an entropy parameter $H \in [0,1]$ of problem domains that measures program diversity, and deduce from this upper bounds on code reuse and the scale of components with which we may work. For "low entropy" domains with H near 0, programs are highly similar to one another and the domain is amenable to the Component-Based Software Engineering (CBSE) dream of programming by composing large-scale components. For problem domains with H near 1, programs require substantial quantities of new code, with only a modest proportion of an application comprised of reused, small-scale components. Preliminary empirical results from Unix platforms support some of the predictions of our model.

1 Introduction and Overview

Software reuse offers the hope that software construction can be made easier by systematic reuse of well-engineered components. In practice reuse has been found to improve productivity and reduce defects [3, 12, 16, 23, 24]. But what of the limits of reuse — will large-scale reuse make software construction easier? Thinking here is varied, but for the sake of argument let me artificially divide the opinions into two competing hypotheses. First the more enthusiastic end of the spectrum, which I associate with the Component-Based Software Engineering (CBSE) movement.

Hypothesis 1 (Strong reuse). *Large-scale reuse will allow mass-production of software, with applications being assembled by composing large, pre-existing components. The activity of programming will consist primarily of choosing appropriate components from libraries, adapting and connecting them.*

Strong reuse is thought to thrive in problem domains with great concentration of effort and similarity of purpose, i.e., many people writing similar software whose requirements show only minor variation. However, the question of whether strong reuse can succeed for software construction considered globally, across disciplines and organizations, remains uncertain. A more cautious view of reuse is the following.

Hypothesis 2 (Weak reuse). *Large-scale reuse will offer useful reductions in the effort of implementing software, but these savings will be a fraction of the code required for large projects. Nontrivial projects will always require the creation of substantial quantities of new code that cannot be found in existing component libraries.*

Representative of weak reuse thinking is the following prescription for code reuse in well-engineered software from Jeffrey Poulin [24]: up to 85% of code ought be reused from libraries, with a remaining 15% custom code, written specifically for the application and having little reuse potential. The percentage of code that may be reused from libraries varies greatly across problem domains, but weak reuse paints a fairly accurate picture of the software landscape of today. Many explanations are proposed for why strong reuse is not happening on a global scale (cf. [9]). A common position in the reuse literature is that if we can only get the right environment in place — the right tools, generalizations, economics, a "culture of reuse" — then reuse of software will soar, with consequent improvements in productivity and software quality.

A contrary view. The perspective developed in this paper suggests that the extent to which reuse can happen is an intrinsic property of a problem domain, and that improving the ability of programmers to find, adapt, deploy, generalize and market components will have only marginal impact on reuse rates if the domain is resistant to reuse. We propose to associate with problem domains an entropy parameter $0 \leq H \leq 1$ measuring the diversity of a problem domain. When $H = 1$, software is extremely diverse and we should expect very little potential for reuse; in fact, we show that the proportion of an application we can draw from libraries approaches zero for large projects. For problem domains with $H < 1$, software is somewhat homogeneous, and with decreasing H comes increasing potential for reuse. The theory we develop suggests that an expected proportion of *at most* $(1-H)$ of an application's code may be reused from libraries, with a remaining proportion H being custom code written specifically for the application. As H nears 0 we enter the strong reuse utopia of "programming

by composing large components." The possibilities of reuse are *strictly limited* by the parameter H, which is an intrinsic property of the problem domain.

We develop this theory by examining our ability to compress or compactify software by the use of libraries. We shall speak throughout this paper of *compressed programs*, by which we mean programs written using libraries, and *uncompressed programs* that are stand-alone and do not refer to library components. The principle tools we employ are information theory and Kolmogorov complexity. Both of these carry subtly different notions of compressibility that we shall have to juggle. The information theory notion deals with compressing objects by identifying patterns that appear frequently and giving them short descriptions — as in English we have taken to saying "car" for "automobile carriage." The Kolmogorov version of compressibility describes our ability to find for a given program a shorter program with the same behaviour, without appealing to how typical that program might be for the problem domain within which we are working. We assume some basic familiarity with information theory as might be found in e.g. [7, Ch. 2] or [20]. The essentials of Kolmogorov complexity are reviewed in Section 3.

Library components and prime numbers. Integers factor into a product of primes; software can be factored into an assembly of components. Library components are the prime numbers of software. This would be a terribly naive thing to say were it not for the many wonderful parallels that turn up:

- There are infinitely many primes; in Section 5.2.1 we prove there are infinitely many components for a problem domain that reduce expected program size (thus guaranteeing employment for library writers.)
- The nth prime is a factor of $\sim \frac{1}{n \ln n}$ of the integers. Theory predicts the nth most frequently used library component has an ideal reuse rate of about $\frac{1}{n \log n \log^+ n}$ (Section 4.2).
- The Erdös-Kac theorem states that the number of factors of an integer tends to a normal distribution; we measure experimental data that suggests a similar theorem might be provable for software components (Figure 4).
- The Prime Number Theorem states that the nth prime is $\sim \log(n \ln n)$ bits long. We show that the ideal configuration for libraries is that the nth most frequently used component is of size $\geq \log n$ and $\leq \frac{1-H}{H} o(n^{\varepsilon})$ for $\varepsilon \geq 0$ (Section 5.2.2).

Reuse and Zipf's Law. It is known that hardware instruction frequencies follow an iconic curve described by George K. Zipf for word use in natural languages [17, 19, 29]. Zipf noted that if words in a natural language are ranked according to use frequency, the frequency of the nth word is about n^{-1}. Zipf-style empirical laws crop up in many fields [25, 22]. Evidence suggests programming language constructs also follow a Zipf-like law [5, 18]. It is natural then to wonder if this result might extend to library components. Our results support this conclusion. Figure 1 shows the reuse counts of subroutines in shared objects on three Unix platforms, clearly showing Zipf-like n^{-1} curves. These results are described in detail in Section 6. The appearance of such curves is not happenstance. In Section 4.2 we argue they are a direct result of programmers

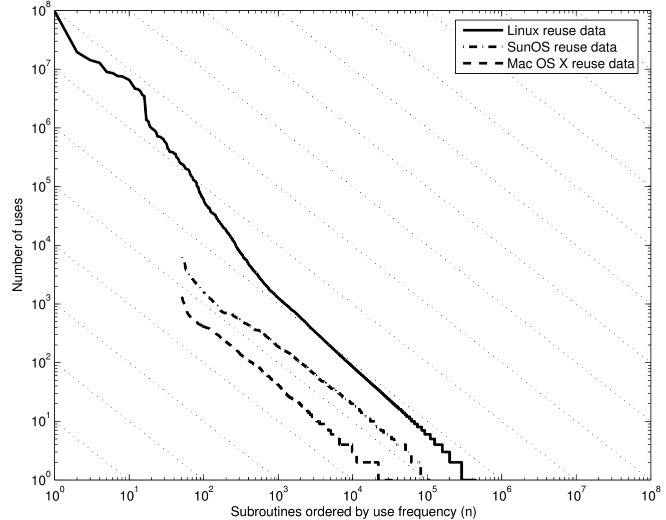


Figure 1. Data collected from shared objects on several unix platforms, showing the number of references to library subroutines. The observed number of references shows good agreement with Zipf-style frequency laws of the form $c \cdot n^{-1}$ (dotted diagonal lines). A detailed explanation of this data is given in Section 6.

trying to write as little code as possible by reusing library subroutines; this drives reuse rates toward a "maximum entropy" configuration, namely a Zipf's law curve.

1.1 Organization

The remainder of this paper is organized as follows. Section 2 introduces an abstract model of software reuse from which we derive our results. In Section 3 we give a brief overview of Kolmogorov complexity. In Section 4 we derive bounds on the rates at which software components may be reused, and give an account for the appearance of Zipf-style empirical laws. Section 5 examines the potential for software reuse as a function of the parameter H. In Section 6 we present some preliminary experimental results, and Section 7 concludes.

2 Modelling library reuse

In this section we propose an abstract model capturing some essential aspects of software reuse within a problem domain. The basic scenario is this: we have a library, possibly many libraries that we collectively consider as one, that contains a great number of software components. These components may be subroutines, architectural skeletons, design patterns, generics, component generators, or whatever form of abstraction we may yet invent; their precise nature is unimportant for the argument. In using a component from the library we achieve some reduction in the size of the program, and perhaps consequently, in the effort required to implement it. Program size serves as a rough lower bound to effort, but it would be a grave error to confuse the two.

2.1 Distribution of programs in a domain

We presume that the projects undertaken by programmers working in a problem domain can be modelled by a proba-

bility distribution on programs. The probability distribution is defined on "uncompressed" programs that do not use any library components. These uncompressed programs can be viewed as *specifications* that programmers set out to realize.

We consider compiled programs modelled by binary strings on the alphabet $\{0,1\}$. We write $\|w\|$ for the length of a string w. Finite programs are countably infinite in number, so we immediately encounter the problem of defining a probability distribution in which the probability of encountering individual programs may be infinitesimal. A rigorous approach would be to employ measure theory, for example Loeb measure, which would allow us to speak of the probability of individual programs. This would require some rather daunting machinery and we instead settle for a more accessible approach similar to that used by [4, 6, 26].

Let $A^{\leq n} = \{w \in \{0,1\}^* \ : \ \|w\| \leq n\}$ denote compiled programs of length at most n bits. We introduce a family of conditional distributions $\{p_{s_0}\}_{s_0 \in \mathbb{N}}$ whose domains consist of programs $\leq s_0$ bits in size, that is,

$$p_{s_0} : A^{\leq s_0} \to \mathbb{R}$$

and satisfying $\sum p_{s_0} = 1$ and $p_{s_0}(w) \geq 0$. The intent is that $p_{s_0}(w)$ gives the probability that someone working in the problem domain will set out to realize the particular (uncompressed) program w, given that w is at most s_0 bits long. For this family of distributions to be compatible with one another we require that $p_{s_0}(w) = p_{s_0+1}(w \mid \|w\| \leq s_0)$, i.e., we can get the distribution on length $\leq s_0$ programs by taking a conditional probability on the distribution for length $s_0 + 1$ programs. We do not presume that such distributions can be effectively described.

In what follows we use the usual notation for expectation with the implied assumption of $s_0 \to \infty$; for example, if $f : \{0,1\}^* \to \mathbb{R}$ maps programs to real numbers, then by $E[f(w)]$ we mean:

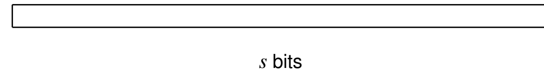
$$E\left[f(w)\right] \equiv \lim_{s_0 \to \infty} \sum_{w : \|w\| \leq s_0} f(w) p_{s_0}(w)$$

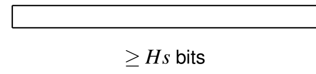
if such a limit should exist. For example, a mean program size $E\left[\|w\|\right]$ may exist for a problem domain, but we do not require nor expect this.

2.2 The entropy parameter H

A key, perhaps defining, feature of a problem domain is that there is similarity of purpose in the programs people write. We do not expect the distribution of programs written in a problem domain to be uniform over all possible programs, but rather concentrated on programs that solve certain classes of problems typical for the domain. We formalize this intuition by introducing a parameter H for problem domains measuring how far their probability distribution departs from uniform. This H is very similar to entropy rate from information theory, and coincides if we are willing to assume programs are drawn from a stationary stochastic process. When $H = 1$ the distribution over programs is uniform, modelling extreme diversity of software, with little opportunity for reuse. For $H < 1$ there is some potential for reuse. In fact as we shall see shortly, we may expect that up to a proportion $1 - H$ of programs may be reused from libraries.

Define the entropy of each distribution p_{s_0} in the standard way





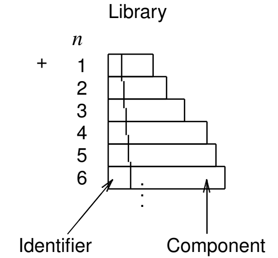


Figure 2. The basic scenario: programmers in a problem domain set out to realize a program that can be represented in s bits when compiled without the use of a library. By using library components, they are able to reduce the size of the compiled program, down to an expected size of $\geq Hs$ bits.

(see, e.g., [7, 20]):

$$H(p_{s_0}) = \sum_{w : \|w\| \leq s_0} -p_{s_0}(w) \log_2 p_{s_0}(w)$$

This is the expected number of bits required to represent a program of size $\leq s_0$ in this domain. We are interested in the limit behaviour of $\frac{1}{|A^{\leq s_0}|} H(p_{s_0})$, akin to the entropy rate of a random process. In general this limit may not exist — there might be oscillations — so we need some weaker notion of limit. We settle for a limsup, which gives an almost sure upper bound on the limit behaviour.

Definition 1 (Entropy parameter). Define the *entropy parameter* H of a problem domain to be the greatest value that $\frac{1}{|A^{\leq s_0}|} H(p_{s_0})$ attains infinitely often as $s_0 \to \infty$:

$$H = \limsup_{s_0 \to \infty} \left(\frac{1}{|A^{\leq s_0}|} H(p_{s_0}) \right)$$

As a consequence of this definition we are guaranteed that $H(p_{s_0}) \leq s_0 H$ almost surely as $s_0 \to \infty$.

We cannot hope to calculate H from first principles except for toy scenarios, but there is hope we might estimate it empirically. We introduce H primarily as a theoretical tool to model problem domains in which people have great similarity of purpose ($H \to 0$) or diffuse interests ($H \to 1$). The main impact of H is the following.

Claim 2.1. *In a problem domain with entropy parameter H, the expected proportion of code that may be reused from a library is at most $1 - H$.*

This is a consequence of the Noiseless Coding Theorem of information theory (e.g., [1, §2.5]), which states that coding random data with entropy H requires (on average) at least H bits. Suppose an uncompressed program has size $s \leq s_0$. We defined H so that $H(p_{s_0}) \leq sH$ almost surely, so we can compress programs to an expected size of at best sH by the Noiseless Coding Theorem. Therefore the expected amount of code saved by use of the library is at most $(1 - H)s$, and it is reasonable to equate this with the amount of code reused from the

library. An immediate implication is that blanket reuse prescriptions such as "effective organizations reuse 70% of their code from libraries" are unrealistic; reuse goals need to be pegged to the problem domain's value of H.

Figure 2 illustrates the scenario we consider in this paper: programmers set out to implement the capabilities of some uncompressed program of length s written without use of a library, drawn from the distribution for the problem domain. A programmer implements the program making use of the library, effectively "compressing" it. The expected size of the compressed program is at least Hs bits, by the previous arguments. The library consists of a set of components, each with an identifier or *codeword* by which they are referred to. We always take programs to be *compiled*, so as not to care about the high compressibility of source representations.

2.2.1 Motifs and the AEP

One question we should like to answer is whether when $H < 1$ there are commonly occurring patterns or "motifs" in programs that we can put in libraries and reuse to compress programs. If we are willing to assume that programs in a problem domain behave as if excerpted from a stationary ergodic source, then the Shannon-McMillan-Breiman theorem (asymptotic equipartition property or AEP) [7, §15.7] ensures that when $H < 1$ there are commonly occurring finite subsequences in programs that can be exploited, and indeed that we can achieve optimal compression of programs merely by having libraries of common instruction sequences. That more complex software components prove necessary in practice suggests the stationary ergodic assumption is too strong, and a weaker ergodic property is needed to account for the emergence of motifs in software when $H < 1$. It is unclear yet exactly what this property might be; in the remainder of this paper we do not assume AEP.

2.3 Libraries maximize entropy

> A truly great computer programmer is lazy, impatient and full of hubris. Laziness drives one to work very hard to avoid future work for a future self. — Larry Wall

Programmers, so we read, are lazy— they write libraries to capture commonly occurring abstractions so they do not have to write them over and over again. The social processes that drive programmers to develop libraries have an interesting theoretical effect. We can view programmers contributing to domain-specific libraries as collectively defining a system for *compressing programs* in that domain. If there is a common pattern, eventually someone will identify it and put it in a library. Since the absence of common patterns in code is implied by high entropy, we propose the following principle.

Principle 1 (Entropy maximization)**.** Programmers develop domain-specific libraries that minimize the amount of frequently rewritten code for the problem domain. This tends to maximize the entropy of compiled programs that use libraries.

As evidence for this principle, we show in Section 6 that the rate at which library components are reused is empirically observed to approach a maximum entropy configuration.[1]

In practice programmers have to strike a balance between the succinctness of their programs and their readability; see, e.g., [11] for an elegant discussion of such tradeoffs. However, we maintain that the drive toward terseness and factoring common patterns is a defining pressure on library development: entropy is essentially a measure of *communication efficiency*, and programmers edge as close to maximum entropy as they can while maintaining source-code understandability.[2]

2.4 The Platonic library

In the early days of computing libraries held a hundred subroutines at most; these days it is common for computers to have a hundred thousand subroutines available for reuse (cf. Section 6). Let us suppose that as time goes on we shall continue to add components to our libraries as we discover useful abstractions and algorithms. Our current libraries might be viewed as a truncated version of some infinite (but countable) library toward which we are slowly converging. It is convenient to pretend that this limit already exists as some infinite "Platonic library" for the problem domain, and that we are merely discovering ever-larger fragments of it, recalling Erdös' book of divine mathematical proofs.[3] Were we granted access to the entire library, we might write software in a very efficient way. We use the Platonic library as a device — a convenient fiction — to reason about how useful finite libraries might be.

Infinite objects need to be treated with care. We shall not assume that some "optimal infinite library" exists that is the best possible such library. Nor shall we assume there is some finite description or computable enumeration of its contents. We merely assume that fragments of the Platonic library give us snapshots of what shall be in our software libraries over time.

2.5 Existence of reuse rates

Numerous metrics have been proposed for measuring reuse. We focus on the *reuse rate* of a component, which we write $\lambda(n)$ and define as the expected rate at which references are made to the nth library component in a compressed program. The units of $\lambda(n)$ are expected references per bit of compiled code. We assume mean reuse rates exist in a problem domain, in the following sense.

Assumption 1. *Let* $\mathrm{Refs}_n(w)$ *count the number of references to the* n^{th} *component in a compressed program* w *of size* $\leq s_0$*. We*

---

[1] Note that Principle 1 is *not* intended to appeal to the maximum entropy principle as advocated by Jaynes, which deals with maintaining uncertainty in inference.

[2] We re-emphasize that we are speaking of the entropy rate of *compiled* programs; source representations are highly compressible to support readability.

[3] A Platonic object is an abstract entity thought to dwell in some realm outside spacetime. Our stance with respect to software libraries echoes mathematical Platonism, that mathematical objects about which we reason exist in some idealized form outside the physical universe (see, e.g., [2]).

assume that

$$E\Big[\mathrm{Refs}_n(w)\ \Big|\ \|w\| = s\Big] \ \sim\ \lambda(n)s + o(s) \quad \text{as } s_0 \to \infty \qquad (1)$$

where $o(s)$ denotes some error term growing asymptotically slower than s.

We unfortunately do not have a good sense of how to go from the problem domain's distributions p_{s_0} on *uncompressed* programs to rates of components in *compressed* programs; this is tied up with the ergodic process issues mentioned in Section 2.2. We dodge the issue by simply *assuming* that the mean rates $\lambda(n)$ exist. This is not a demanding assumption; many sensible random process models would imply Assumption 1, for example modelling component uses as a renewal process (see, e.g., [27, §3]).[4]

2.6 Ordering of library components

For convenience we shall suppose the library components are arranged in decreasing order of expected reuse rate in the problem domain: that is,

$$\lambda(n) \geq \lambda(n+1)$$

There are two reasons for this. The first is tidiness, so that when we plot $\lambda(n)$ vs n we see a monotone function and not noise. The asymptotic bounds we derive on $\lambda(n)$ do not rely on this ordering. The second reason is that to derive bounds on how well we might compress programs we need to assign shorter identifiers to more frequently used components. This is easiest to reason about if the Platonic library is sorted by use frequency.[5]

3 Kolmogorov Complexity

Kolmogorov complexity, also known as Algorithmic Information Theory, was founded in the 1960s by R. Solomonoff, G. Chaitin, and A.N. Kolmogorov. We shall only make use of some basic facts; for a more thorough introduction the survey article [21] or the book [20] are recommended. The central idea is simple: measure the 'complexity' of an object by the length of the smallest program that generates it. This generalizes to the study of *description systems*, that is, systems by which we define or describe objects, of which programming languages, logics, and descriptive set theory are prominent examples. The source code of a program, for example, describes a program behaviour; a set of axioms describes a class of mathematical structures. In the general case we have some objects we wish to describe, and a description system ϕ that maps from a description w (for us, a program) to objects. The usual situation is to describe an object by exhibiting a program that generates it; in this case we may also provide some inputs to the program, which we shall call *parameters*. The Kolmogorov complexity of an object x in the description system ϕ, relative to a parameter y is defined by:

$$C_\phi(x \mid y) = \min_w\{\|w\| : \phi_w(y) = x\} \qquad (2)$$

In the case where the description system ϕ is a programming language, we may read Eqn. (2) as finding the shortest program that, given input parameter y, outputs x. The parameter y does not contribute to the measured description length $C_\phi(x \mid y)$. Without a parameter we have the simpler case $C_\phi(x) = C_\phi(x \mid \varepsilon)$ where ε is the empty string.

For example, we might choose the programming language Java as our description system; then for some string x, its Kolmogorov complexity $C_{\mathsf{Java}}(x)$ is the length of the shortest program that outputs x. To determine whether use of a library L offers a reduction in program size, we can consider the combination of Java and the library L as a description system itself which we might call $\mathsf{Java} + \mathsf{L}$, and compare $C_{\mathsf{Java}+\mathsf{L}}(x)$ to $C_{\mathsf{Java}}(x)$.

A very useful insight is that the choice of language doesn't much matter.

Fact 3.1 (Invariance [20, §2.1])**.** *There exists a universal machine U such that if ϕ is some effective description system (e.g., a programming language) then there is a constant c such that $C_U(x) \leq C_\phi(x) + c$ for any x.*

That is, the universal machine U is optimal up to a constant factor. For this reason the subscript U can be dropped and one can write $C(x)$ for the Kolmogorov complexity of x, knowing it is only defined up to some constant factor.[6]

Some strings have very short descriptions: a string of a trillion zeros may be produced by a short program. Others require descriptions as long as the strings themselves, for instance a million digit binary string obtained from a physical random bit generation device.[7] A recurrent theme in Kolmogorov complexity is that there are never enough descriptions to go around so as to give short descriptions to most objects. In the case where both the objects and their descriptions are binary strings, we have the following well-known result that the probability we can save more than a constant number of bits in compressing randomly selected strings is zero.

Fact 3.2 (Incompressibility [20, §2.2])**.** *Suppose $g : \mathbb{N} \to \mathbb{N}$ is an integer function with $g(n) > 0$ and $g \in \omega(1)$, that is, $\lim_{n\to\infty} g(n) = \infty$. Let x be a string chosen* uniformly at random. *Then almost surely:*

$$C_\phi(x) \geq \|x\| - g\big(\|x\|\big) \qquad (3)$$

Fact 3.2 implies, for example, that one cannot devise a coding system that compresses strings by even $\log\log n$ or $\alpha^{-1}(n,n)$ (inverse Ackermann) bits with nonzero probability. The proof of Fact 3.2 uses counting arguments only, with no appeal to computability of the description system.[8] Therefore the in-

[4] For readers familiar with coding theory we forestall confusion by mentioning that the rates $\lambda(n)$ are not the same as the usual notion of probabilities over countable alphabets. The rates $\lambda(n)$ are drawn from compressed programs and so already incorporate code lengths.

[5] Jeremiah Willcock made the useful suggestion that we may regard the Platonic library as containing already every possible component, and the only question is the order in which they are placed.

[6] There is an easy way to see why this is true: if ϕ is a programming language, then we can write a ϕ-interpreter for the universal machine U. We can then take any program for ϕ, prepend the interpreter, and it becomes a U-program. The constant mentioned reflects the size of such an interpreter.

[7] Unless you are rather lucky.

[8] There are $2^{n-g(n)+1} - 1$ descriptions of length at most

equality (3) applies to *any* description system ϕ, even description systems that are not computable. For example, Fact 3.2 even applies if we permit ourselves to use an infinite, not computably enumerable library as we described in Section 2.4. However, it does not apply in the case where there is a nonuniform distribution, as in problem domains where $H < 1$.

In the remainder of this paper we shall assume compiled programs are incompressible in the sense of Fact 3.2.

Proposition 3.1. *Compiled C programs on existing major architectures are almost surely Kolmogorov incompressible.*

Note that "almost surely *Kolmogorov* incompressible" does not imply anything about the compressibility of *typical* compiled programs for a problem domain. Rather, it means that if one chooses a valid compiled program uniformly at random, with probability 1 it cannot be replaced by a shorter program with the same behaviour. In subsequent sections we investigate problem domains where there *is* a nonuniform distribution on programs, i.e., $H < 1$, where the situation is rosier.

We sketch a proof of Proposition 3.1, showing that the number of distinct behaviours described by compiled programs of s bits grows as $\sim 2^s$ on current machines, which implies compiled programs are almost surely (Kolmogorov) incompressible. The C language has the useful ability to incorporate chunks of binary data in a program. For example, the binary string $z = 0110100111011010$ may be encoded by the C declaration

$$\mathsf{unsigned\ char\ z}[2] = \{\mathsf{0x69}, \mathsf{0xda}\};$$

Moreover, such arrays are laid out as contiguous binary data in the compiled program, so that a binary string of length m bytes requires exactly m bytes in the compiled program. We can package such an array with a short program of constant size that reads the binary string from memory and outputs it to the console. Every binary string of m bytes may be encoded by such a compiled program of size at most $c+m$ bytes, where c is a constant representing the overhead of a read-print loop. Every such program yields a unique behaviour, so the number of distinct behaviours of compiled programs of s bits is $\sim 2^s$. We can then adapt the argument used to prove Fact 3.2, replacing strings by compiled programs, which shows compiled C programs are almost surely incompressible.

Note that uncompiled programs are *highly* compressible. For example, C language source code may not contain certain bytes (e.g., control characters) such as the null character $0x00$. This means they can be compressed by a factor of (at least) $\frac{1}{256} \sim 0.39\%$. Restricting our attention to *compiled* programs is crucial.[9]

$n - g(n)$, and $2^{n+1} - 1$ strings of length at most n. Therefore the fraction of strings compressible by $g(n)$ bits is at most $\frac{2^{n-g(n)+1}-1}{2^{n+1}-1}$, which behaves in the limit as $2^{-g(n)}$. If $g \in w(1)$ this value vanishes as $n \to \infty$, so $C_\phi(x) \geq \|x\| - g(\|x\|)$ almost surely.

[9] An alternative would be to deal with *indices* of programs in the usual sense of computability theory, where we equate a program with its position in some effective enumeration of valid source-language programs. However, working with compiled programs has the additional benefit of brushing aside issues such as identifier lengths in source code, which tend to be unnecessarily long to aid readability.

4 A bound on reuse rates

In this section we derive a bound on the reuse rate $\lambda(n)$ at which the nth library component is reused in 'compressed' programs written with use of a library.

4.1 Coding of references

We need some rudimentary accounting of what we gain and lose by use of the library: we save some by using a library component, at the cost of having to refer to it. Let us first consider the cost of referring to components.

We presume that unique identifiers are assigned to library components; we call these *codewords*. Let $c(n)$ be the binary codeword for the nth library component, and $\|c(n)\|$ its length. Optimal strategies such as Shannon-Fano or Huffman codes assign shortest codewords to the most frequently needed components. Since our library is sorted in order of use frequency (Section 2.6), we may presume that $\|c(n)\| \leq \|c(n+1)\|$, i.e., codeword lengths are nondecreasing as we go down the list of components.

In what follows we want to make asymptotic arguments, and fixing an identifier size (e.g., 64 bits) would lead to wildly wrong conclusions.[10] Instead we require that the identifier size grows with the number of components, albeit slowly. That $\|c(n)\| \geq \log_2 n$ follows from the pigeonhole principle. Having identifiers of length only $\log_2 n$ leads to difficulties, because they are not *uniquely decodable*. That is, if I am presented with a string of such identifiers I have no way to tell where one identifier stops and the next starts. (This does not arise in current architectures because of fixed word size, but as we said, care is needed in asymptotic arguments). A more accurate requirement is the following, which draws on Kraft's inequality that uniquely decodable codes must satisfy $\sum_{n=1}^{\infty} 2^{-\|c(n)\|} \leq 1$.

Proposition 4.1. *For identifiers to be uniquely decodable,*

$$\|c(n)\| \geq \log^+ n$$

where $\log^+ n = \log n + \log\log n + \log\log\log n + \cdots$ *and the sum is taken only over the positive real terms.*

We omit the proof; see e.g., [26, §2.2.4] or [20, §1.11.2] (in particular problem 1.11.13).

4.2 Derivation of reuse rate bound

We now derive an asymptotic upper bound on the rates $\lambda(n)$ at which library components may be reused. We do this under the assumption that each time a library component is used in a program, the same identifier is used to refer to it, i.e., there is no *recoding of identifiers*.[11] Our argument follows standard

[10] If we fix memory addresses to be representable in 64 bits, then the time to search an acyclic linked list is $O(1)$ since there are at most 2^{64} steps the algorithm must go through.

[11] There are two reasons for this assumption. (1) On the architectures from which we collect empirical data, there is no recoding of identifiers in programs. (2) The reason one might want to recode identifiers is to save space by introducing shorter aliases for components for use within the program, after the initial reference. However, this only saves space if a

lines [25] but adapted to coding of library references under the model laid out in Section 2.

Theorem 4.1. *Without recoding of identifiers, the asymptotic reuse rates $\lambda(n)$ must satisfy $\lambda(n) \prec (n \log n \log^+ n)^{-1}$.*

PROOF. We count the size of the references to library components within compressed programs (i.e., those written with use of a library). Consider programs of length at most s. As $s \to \infty$, the expected number of occurrences of the nth component tends to $\lambda(n)s + o(s)$ under Assumption 1. Referring to the nth component requires at least $\log^+ n$ bits (Proposition 4.1). We need only consider components whose identifier length is less than s, since identifiers longer than the program would not fit. Therefore we consider only up to component number 2^s since $\log^+ 2^s \geq s$.

The expected total size of all the references to components is then at least:

$$\sum_{n=1}^{2^s} \underbrace{\big(\lambda(n)s + o(s)\big)}_{\#\text{ refs}} \underbrace{\log^+ n}_{\text{ref size}}$$

The references to components are contained within the program, and therefore their total size must be less than s, the size of the program. Therefore we have an inequality:[12]

$$\sum_{n=1}^{2^s} \big(\lambda(n)s + o(s)\big) \log^+ n \leq s \tag{4}$$

Dividing through by s and taking the limit as $s \to \infty$,

$$\lim_{s\to\infty} \sum_{n=1}^{2^s} \frac{1}{s} \big(\lambda(n)s + o(s)\big) \log^+ n \leq 1 \tag{5}$$

Since $\lim_{s\to\infty} \frac{1}{s} o(s) = 0$ by definition,[13]

$$\sum_{n=1}^{\infty} \lambda(n) \log^+ n \leq 1 \tag{6}$$

We now consider conditions under which this sum converges. (Section A.1 summarizes the asymptotic notations used here.) We argue using Proposition A.1, using a diverging series to bound the terms of Eqn. (6). The simple argument is to note that the harmonic series diverges, and therefore the terms of Eqn. (6) must grow slower than this, so $\lambda(n) \log^+ n \prec \frac{1}{n}$, or $\lambda(n) \prec \frac{1}{n \log^+ n}$. However, this bound is quite loose. A more slowly diverging series is $\sum_n \frac{1}{n \log n}$. Using this,

$$\lambda(n) \log^+ n \prec \frac{1}{n \log n}$$

or,

$$\boxed{\lambda(n) \prec \frac{1}{n \log n \log^+ n}} \tag{7}$$

This completes the proof. □ □

The bound of Theorem 4.1 is not tight. No tightest bound is possible using this line of argument since there is no slowest diverging sequence with which to bound a convergent sequence, a classical result due to Niels Abel. However, the bound is tight to within a factor n^{ε} for any $\varepsilon > 0$.

Entropy maximization and Zipf's Law. Theorem 4.1 provides an upper bound on $\lambda(n)$, but it could well be the case that $\lambda(n) \sim \frac{1}{n^3}$, for example. Why do the curves we see in practice (e.g., Figure 1) hug the bound of Theorem 4.1? We believe the answer to why we observe $\lambda(n) \approx \frac{1}{n}$ is due to the tendency of libraries to evolve so that programmers can write as little code as possible, which in turn implies evolution toward maximum entropy in compiled code (Principle 1). The entropy rate of component references is maximized when $\lambda(n) \approx \frac{1}{n}$ (see, e.g., [13]).

5 Reuse potential

In the following sections we consider the possibilities of code reuse in two cases: (1) when $H = 1$ and we have a uniform distribution on programs; (2) when $0 < H < 1$ and we have some degree of compressibility in the problem domain. The case $H = 0$ is left for future work.

5.1 The uniform case: $H = 1$

The uniform case of $H = 1$, in which every program is equally likely to be implemented, reduces the scenario to classical Kolmogorov complexity with a uniform distribution on programs. It has some surprising properties that suggest $H = 1$ to be an unlikely scenario for real problem domains.

Our first result concerns the number of library components we might expect to use in a program. Let $N(s)$ be a random variable indicating for a program of uncompressed size s the number of components whose use reduces program size. Surprisingly, as program size increases the expected number of components that reduce program size is bounded above by a constant.

Theorem 5.1. *If $H = 1$ there exists a constant n_{crit} independent of program size s such that $N(s) \leq n_{\mathsf{crit}}$ almost surely.*

PROOF. Suppose each component used saved at least 1 bit. If $\lim_{s\to\infty} E[N(s)]$ were unbounded, use of the library could compress random programs by an unbounded amount, contradicting incompressibility (Fact 3.2). □ □

This has a simple corollary concerning the potential for code reuse.

component is more likely to be used again given it is used once. While this is intuitively true of real programs, it is false under a maximum entropy assumption (Section 2.3): in an encoding that maximizes entropy, the sequence of identifiers in a program behaves statistically as if independent and identically distributed.

[12] Inequality (4) becomes an equation if we consider programs to consist solely of a sequence of component references, with no control flow or other distractions. This is possible by building components and programs from combinators, which can be made self-delimiting [20, §3.2]. This provides a theoretically elegant framework, if not entirely intuitive.

[13] Recall that $f \in o(g)$ means $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$.

Corollary 5.1. *When $H = 1$ the expected proportion of a program that can be reused from libraries tends to zero as program size increases.*

Because of these results, the case $H = 1$ is somewhat uninteresting and does not seem to model real life, where we know libraries are useful and let us reduce the size of programs. In the next sections we examine the more interesting case of $0 < H < 1$, where we can compress programs, even ones that are (Kolmogorov) incompressible, by use of a library.

5.2 The nonuniform case: $0 < H < 1$

More interesting than the uniform case is the situation when $0 < H < 1$, which implies a nonuniform distribution on programs. This models problem domains that have some potential for code reuse, and libraries are of central importance in reducing program size. Recall from Section 2.2 that we can expect to compress programs in such domains from uncompressed size s to at best Hs by use of a library. A standard result from information theory can be adapted to show this bound is achievable, at least in a theoretical sense.

Claim 5.1. *There exists a library with which uncompressed programs of size s can be compressed to expected size $\sim Hs$.*

The proof of this is not particularly illustrative and we banish it to a footnote.[14] The gist is to place every possible program into the library as a "component," but ordering them so that the most likely programs for the problem domain come soonest in the library order and thus are assigned the shortest codewords. This is a wildly impractical construction but demonstrates the claim. In practice we decompose software into reusable chunks that we put in libraries; that reusable chunks exist suggests an ergodic property (see Section 2.2.1).

Unlike the situation of $H = 1$ where the number of components useful for a program was at most a constant, when $0 < H < 1$ we have a much more pleasing situation: the number of useful components increases steadily as we increase program size.

[14] *Proof.* We first describe an encoding that compresses programs to achieve an expected size Hs, and then explain how to construct the library. Recall the Shannon-Fano code [20, §1.11] allows a finite distribution with entropy H to be coded so that the expected codeword length is $\leq H+1$. We adapt this as follows. For each $s_0 \in \mathbb{N}$, we produce a Shannon-Fano codebook for all programs of length $\leq s_0$ achieving average codeword size $\leq H(p_{s_0})+1$ for the distribution p_{s_0} (Section 2.2). By definition $H(p_{s_0}) \leq Hs$ almost surely, so this achieves a compression ratio of H almost surely for each s_0 as $s_0 \to \infty$. To combine all the codebooks into one, we preface a compressed program with an encoding of its uncompressed length, which we use to select the appropriate codebook. This can be done by adding to each codeword $c + 2\log s$ bits for some constant c, which is negligible with respect to Hs when $H > 0$. Therefore this encoding achieves expected program size $\sim Hs$. We use the codebook as the library: each component identifier is a Shannon-Fano code, each component is a program. Note that the reuse rates vanish for this construction, i.e., $\lambda(n) \to 0$ as $s_0 \to \infty$, and so the bound of Theorem 4.1 is trivially satisfied. □

5.2.1 The incompleteness of libraries

Under reasonable assumptions we prove that no finite library can be complete: there are always more components we can add to the library that will allow us increase reuse and make programs shorter. To make this work we need to settle a subtle interplay between the Kolmogorov complexity notion of compressibility (there is a shorter program doing the same thing) and the information theoretic notion of compressibility (low entropy over an *ensemble* of programs). Now because we defined probability distributions on programs (rather than behaviours), we run into the possibility that the probability distribution might weight heavily programs that are *Kolmogorov* compressible, i.e., the distribution might prefer programs w with $\|w\| >> C(w)$. For example, a problem domain might have programs that are usually compressible to half their size not because the probability distribution focuses on a particular class of problems, but because we artificially defined p_{s_0} to select only those programs that are twice as large as they might be (for example, we might pad every likely program with many nop instructions.) To avoid this difficulty we require the distributions be *honest* in the following sense.

Definition 2 (Honesty)**.** We say the distributions p_{s_0} for a problem domain are *honest* if the programs are Kolmogorov incompressible. Specifically,

$$E\left[\frac{C(w)}{\|w\|}\right] \to 1 \quad \text{as } s_0 \to \infty \tag{8}$$

where the expectation is taken over the distributions p_{s_0}. This requires that the probability distribution does not artificially prefer verbose programs with $\|w\| >> C(w)$.

If the distribution for a problem domain is honest and has $H < 1$, the programs are expected to be *information-theoretically compressible* by use of a library, but not *Kolmogorov compressible*. In other words, our ability to compress programs is due to a "focus" on a class of problems of interest to the domain, not just an artificial selection of overly-verbose programs.

Inspired by Euclid's proof that there are infinitely many primes, with the honesty assumption we can prove there are infinitely many reusable software components that make programs shorter.

First we need two smaller pieces of the puzzle.

Lemma 5.1. *If $H > 0$ then for any finite k, $Pr(\|w\| \leq k) \to 0$ as $s_0 \to \infty$.*

Proof. We know from definition of H that $H(p_{s_0}) = Hs_0$ infinitely many times as $s_0 \to \infty$ (Section 2.2). Consider how probability must be distributed among programs of different lengths to account for this much entropy. We try to account for as much entropy as we can by short programs, setting a uniform distribution $p(w) = \frac{1}{2^{Hs_0}}$ on the first 2^{Hs_0} programs—this is the fewest number of programs that would produce this much entropy. To programs of length $\leq k$ we can account for

$$\sum_{i=0}^{k} 2^i \cdot \left(-\frac{1}{2^{Hs_0}} \log \frac{1}{2^{Hs_0}}\right) \sim 2^{k+1-Hs_0} Hs_0$$

bits of entropy. But as $s_0 \to \infty$, $2^{k+1-Hs_0} Hs_0 \to 0$ so we can account for none of the entropy by programs of length $\leq k$. Therefore $Pr(\|w\| \leq k) \to 0$ as $s_0 \to \infty$. □

Lemma 5.2. *If $H > 0$ then $E\left[\frac{1}{\|w\|}\right] \to 0$ as $s_0 \to \infty$.*

PROOF. Suppose $E\left[\frac{1}{\|w\|}\right] = c$ for some $c > 0$. Then there would be a finite probability that $\|w\| \leq c^{-1}$ as $s_0 \to \infty$, contradicting Lemma 5.1. □

Now we are ready for the main theorem, which proves no finite library can be "complete" in the sense of achieving a compression ratio of H when $0 < H < 1$.

Theorem 5.2 (Library Incompleteness)**.** *If a problem domain has $0 < H < 1$ and honest distributions (Defn. 2), no finite library can achieve an asymptotic compression ratio of H.*

PROOF. Suppose a finite library of components achieves a compression factor $1-\varepsilon$, with optimal compression when $1-\varepsilon = H$. Call the programming language ϕ and the library L. We can write an interpreter for ϕ that incorporates the library L; since the library is finite this is a finite program. We call the resulting machine model $\phi + L$. Consider Kolmogorov complexity for this machine, writing $C_{\phi+L}(w)$ for the size of the smallest ϕ-program using L that has the same behaviour as w. Saying the machine $\phi + L$ achieves the compression factor $1-\varepsilon$ implies

$$E\left[\frac{C_{\phi+L}(w)}{\|w\|}\right] = 1-\varepsilon \tag{9}$$

From the invariance theorem of Kolmogorov complexity (Fact 3.1) we have that there exists a constant c such that

$$C(w) \leq C_{\phi+L}(w) + c \tag{10}$$

for every program w. Dividing through by $\|w\|$ and taking expectation,

$$E\left[\frac{C(w)}{\|w\|}\right] \leq \underbrace{E\left[\frac{C_{\phi+L}(w)}{\|w\|}\right]}_{=1-\varepsilon} + E\left[\frac{c}{\|w\|}\right] \tag{11}$$

From honesty $E\left[\frac{C(w)}{\|w\|}\right] \to 1$, and from Lemma 5.2 we have $E\left[\frac{c}{\|w\|}\right] \to 0$. Therefore (11) is, in the limit as $s_0 \to \infty$:

$$1 \leq (1-\varepsilon) + 0$$

For this inequality to hold, $\varepsilon \to 0$ for any finite library. Therefore no finite library can achieve an asymptotic compression ratio < 1 when the distributions are honest. □ □

Claim 5.1 showed that an infinite library can achieve expected size $\sim Hs$; Theorem 5.2 shows that no finite library can. Therefore only infinite libraries can compress programs of size s to expected size Hs. However, this is an asymptotic argument; if we restrict ourselves to programs of size $\leq s_0$ for some fixed s_0, finite libraries can approach a compression ratio of Hs by including more and more components. Doug Gregor suggested calling Theorem 5.2 the *Full Employment Theorem for Library Writers*, after Andrew Appel's boon to compiler writers. Theorem 5.2 has a straightforward implication: no finite library can be complete; there are always more useful components to add. In practice we have a tradeoff between the utility of larger libraries and the economic cost of producing them; this suggests the importance of designing libraries for extensibility.

A minor change to the above proof yields a similar but slightly stronger result.

Corollary 5.2. *If a problem domain has $0 < H < 1$ and honest distributions, no* computably enumerable *library can achieve a compression ratio of H.*

PROOF. Repeat the proof of Theorem 5.2, replacing "finite library" with "c.e. library." In particular the choice of a c.e. library guarantees that the interpreter for $\phi + L$ is a finite program: whenever a library subroutine is required, it can be generated from the program enumerating the library. □ □

We may casually equate "not computably enumerable" with "requires human creativity." Corollary 5.2 indicates that the process of discovering new and useful library components is not a process that can can be fully automated.

5.2.2 Size of library components.

We now consider how big library components might be. If we want to achieve the strong reuse vision of "programming by wiring together large components," this suggests that components ought to be quite large compared to the wiring. The following theorem sheds light on the conditions when this is possible.

Let $S(n)$ denote the expected amount of code (in bits) saved per use of the nth component. We consider the case when $\lambda(n) \sim \frac{1}{n\|c(n)\|f(n)}$, where $\|c(n)\|$ is the codeword (identifier) length, and $f(n)$ is a function $f \in o(n^\varepsilon)$ for $\varepsilon > 0$ that ensures convergence (cf. Section 4.2). This coincides with a Zipf-style law as observed in practice (Figure 1).

Theorem 5.3. *If a library achieves a compression factor of $H > 0$ in an honest problem domain, then $S(n) \sim \frac{1-H}{H} \cdot o(n^\varepsilon)$ for any $\varepsilon > 0$.*

PROOF. Summing over all components, the total code saved is:

$$\sum_{n=1}^{\infty} \underbrace{(\lambda(n)Hs + o(Hs))}_{\text{expected \# uses}} \cdot \underbrace{S(n)}_{\text{savings per use}} = \underbrace{(1-H)s}_{\text{total savings}} \tag{12}$$

Dividing through by Hs and taking the limit as $s \to \infty$, and substituting $\lambda(n) \sim \frac{1}{n\|c(n)\|f(n)}$,

$$\sum_{n=1}^{\infty} \frac{1}{n\|c(n)\|f(n)} S(n) = \frac{1-H}{H}$$

Now if $S(n) \sim n^a$ for some constant $a > 0$ then the sum would diverge. Therefore $S(n)$ is not polynomial in n; in fact for the sum to converge we must have $S(n) \prec f(n)$ which means $S(n)$ behaves asymptotically as

$$S(n) \sim \frac{1-H}{H} o(n^\varepsilon)$$

where $o(n^\varepsilon)$ denotes some subpolynomial function. □ □

See also Figure 3. Note that if the components in the library are unique, then $S(n) \geq \log n$ by pigeonhole.

Strong reuse? The interpretation of Theorem 5.3 is fairly intuitive. Roughly it says the savings we can expect per compo-

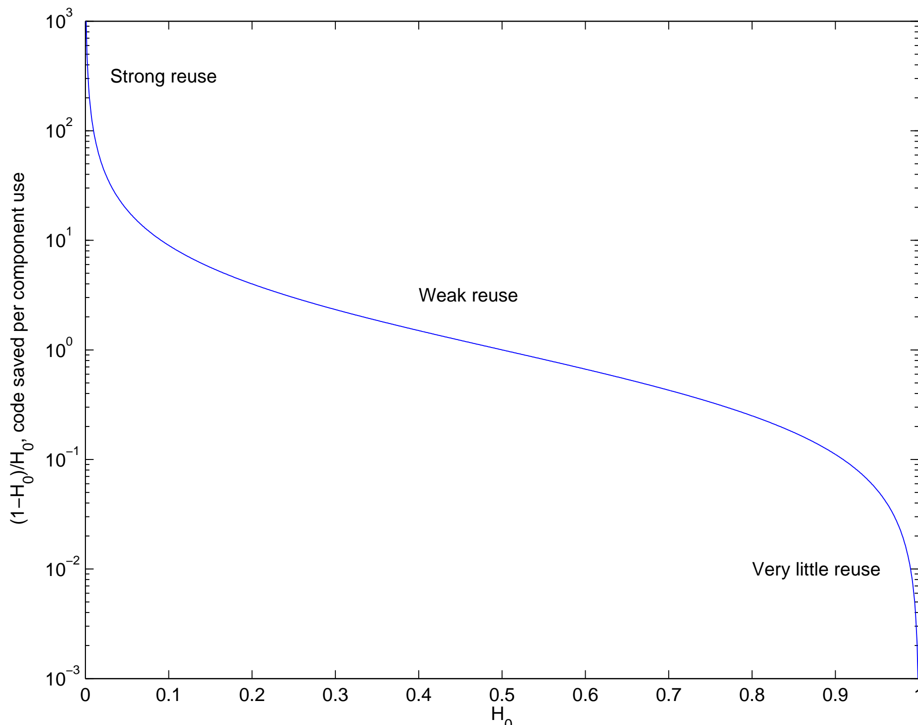


Figure 3. Plot of $\frac{1-H}{H}$ versus H, indicating how much code is saved, proportionately, per component use. When $H \to 1$ there is almost no reuse; $H \to 0$ coincides with the "strong reuse" ideal of wiring together large components. In between is weak reuse, with moderate amounts of code drawn from libraries.

nent are linear in the size of the component identifier. Which is to say, we should expect savings for the nth component to grow roughly as $\log^+ n$. This is consistent with findings in the reuse literature that *small* components are much more likely to be reused. The important factor here is the multiplier $\frac{1-H}{H}$. As $H \to 0$, this multiplier becomes arbitrarily large. This suggests that "strong reuse" (Section 1) corresponds to the region $H \to 0$. For example, if programs in a problem domain are thought to be solvable by wiring together components that are (say) 1000 times bigger than the wiring itself, this suggests $\frac{1-H}{H} \approx 10^5$ or $H \approx 0.001$. The key result is that whether one is able to achieve strong reuse depends critically on the parameter H — which measures how much diversity there is in the problem domain.

6 Experimental data collection

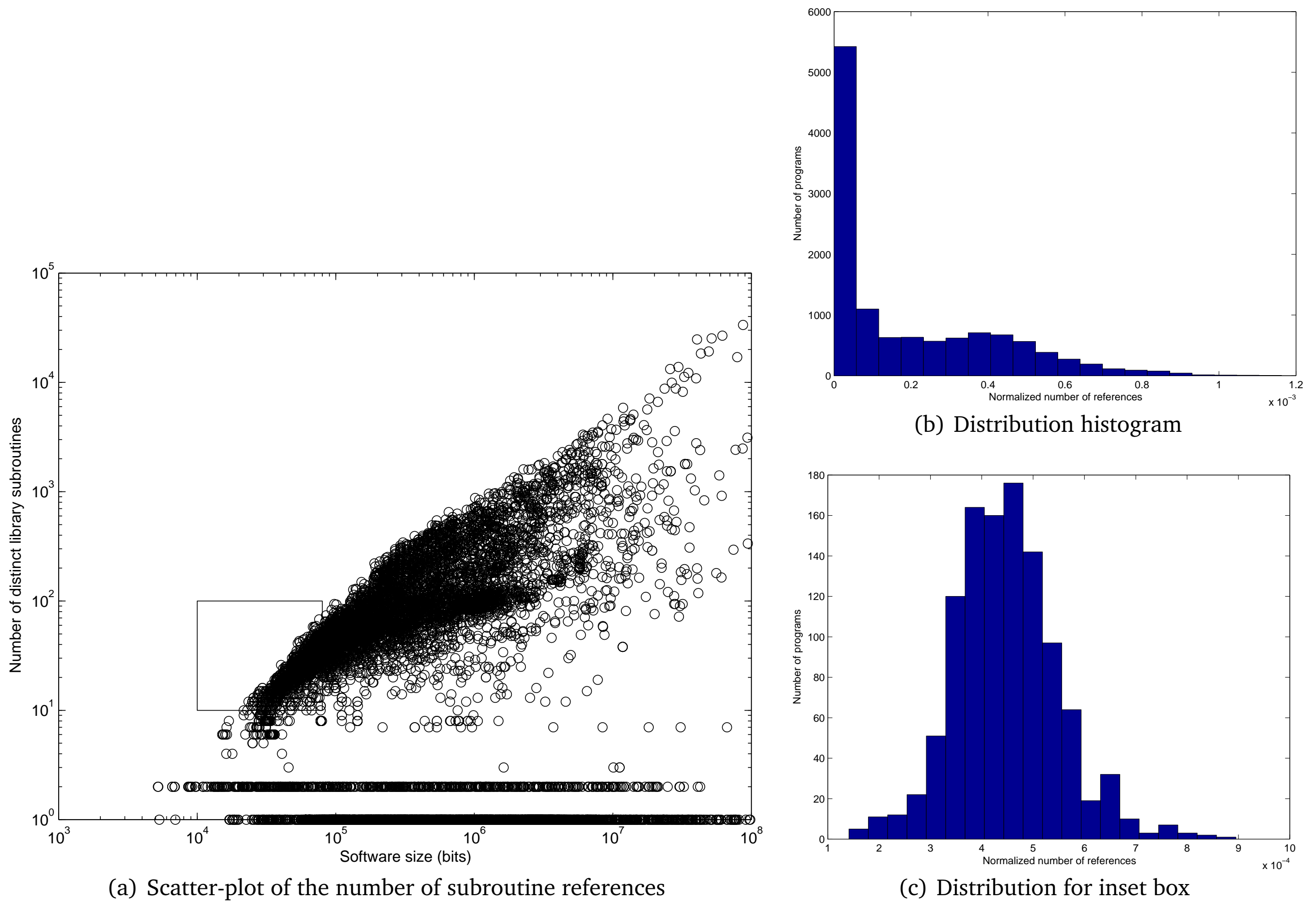
Preliminary empirical data was collected from three large Unix installations. The problem domain is not particularly well-defined, but is rather "the mishmash of things one wants to do on a typical research Unix machine." On the SunOS and Mac OS X machines we located every shared object and used the unix commands nm or objdump to obtain a listing of the relocatable symbols (i.e., references to subroutines in shared libraries). For the Linux machine, a more sophisticated approach was used that involved disassembling every executable object and decoding the PLT and GOT tables for shared library calls. For this reason the Linux data is much more fine-grained and reliable; for example, our data set for Linux includes the frequencies of all the x86 machine instructions, in addition to almost a half-million subroutines.

Operating System	# Objects	# Components
Linux (SuSE)	12136	455716
SunOS	23774	110306
Mac OS X	2334	37677

We counted the number of references to each component, sorted these by frequency, and this data is plotted in Figure 1. The observed counts match nicely the asymptotic prediction made in Section 4.2 (the family of curves cn^{-1} is shown as dotted lines). To account for machine instructions, which are not included in the tally for the Mac OS X and SunOS machines but constitute by far the most frequently occurring software components, we started numbering the components for these machines at $n = 50$. Without this adjustment the rates have a characteristic "flat top" and then rapidly converge to n^{-1} lines; this is an artifact of the log-log scale.

The pronounced "steps" in the data for large n occur because there are many rarely-used subroutines with only a few references; this is typical of such plots (see, e.g., [25]).

Another prediction that may follow from our model is that the number of distinct components used in a program should approach a normal distribution: under maximum entropy conditions the use of components is statistically independent, and so the central limit theorem applies. This is reminiscent of the Erdös-Kac theorem [8] that the number of prime factors of integers converges to a normal distribution. Figure 4 shows some preliminary results that support this result, drawn from



(a) Scatter-plot of the number of subroutine references

(b) Distribution histogram

(c) Distribution for inset box

Figure 4. Data suggesting a library analogue of the Erdös-Kac theorem. (a) A scatter-plot showing the number of distinct library subroutines used vs. software size for the Linux RPM data. (b) Histogram for the number of references, normalized (see text). (c) Histogram only for the inset box of (a), illustrating an Erdös-Kac-style normal distribution for the number of components used in software. Such plots might provide a useful tool for assessing the extent of reuse vs. ideal predictions from a model.

the SuSE Linux data. The number of component references have been normalized by an estimated variance of $\sigma^2 = cs^2$ where s is the program size. Subfigure (c) shows a suggestively shaped distribution for the inset box of (a), a region where there is good "saturation" of the problem domain with programs.

Our preliminary data demonstrates a Heaps' style law for vocabulary growth [15, §7.5]: the number of unique components encountered in examining the first s bytes of the corpus grows roughly as a power law s^α with $\alpha \approx 0.8$. We have not found a satisfactory theoretical explanation.

7 Conclusion

We have developed a theoretical model of reuse libraries that provides good agreement, we feel, with our intuitions, the literature, and the preliminary experimental data we have collected on reuse on Unix machines. Much of what we have done has served to emphasize the importance of this one quantity, H, the entropy rate we associate with a problem domain. It determines if we can have strong reuse ($H \to 0$), or whether we can have weak reuse ($0 < H < 1$), and how much code we might be able to reuse from libraries: at most $1 - H$.

We have shown that libraries allow us to "compress the incompressible," reducing the size of programs that are Kolmogorov-incompressible by taking advantage of the commonality exhibited by programs within a problem domain. We have also shown that libraries are essentially incomplete, and there will always be room for more useful components in any problem domain.

The arguments made here are quite general and might adapt easily to other description systems, for example, the reuse of abstractions, lemmas and theorems in mathematical proofs.

8 Acknowledgments

This paper benefited immeasurably from discussions with my colleagues at Indiana University Bloomington. In particular I thank Andrew Lumsdaine, Chris Mueller, Jeremy Siek, Jeremiah Willcock, Douglas Gregor, Matthew Liggett, Kyle Ross, and Brian Barrett for their valuable suggestions. I thank Harald Hammerström for letting me disappear with his copy of Li and Vitányi [20] for most of a year.

9 References

[1] Robert Ash. *Information Theory*. John Wiley & Sons, New York, 3 edition, 1967.

[2] Mark Balaguer. Platonism in Metaphysics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2004.

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, 1996.

[4] S. Ben-David, B. Chor, O. Goldreich, and M. Luby. On the theory of average case complexity. *Journal of Computer and System Sciences*, 44(2):193–219, April 1992.

[5] Daniel M. Berry. A new methodology for generating test cases for a programming language compiler. *SIGPLAN Not.*, 18(2):46–56, 1983.

[6] Kevin J. Compton. 0–1 laws in logic and combinatorics. In I. Rival, editor, *Proceedings NATO Advanced Study Institute on Algorithms and Order*, pages 353–383, Dordrecht, 1988. Reidel.

[7] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley & Sons, 1991.

[8] P. Erdös and M. Kac. The Gaussian law of errors in the theory of additive number theoretic functions. *Amer. J. Math.*, 62:738–742, 1940.

[9] William B. Frakes and Christopher J. Fox. Quality improvement using a software reuse failure modes model. *IEEE Transactions on Software Engineering*, 22(4):274–279, April 1996.

[10] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, USA, second edition, 1994.

[11] T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the HCI'89 Conference on People and Computers V*, Cognitive Ergonomics, pages 443–460, 1989.

[12] M. L. Griss. Software reuse: From library to factory. *IBM Systems Journal*, 32(4):548–566, 1993.

[13] P. Harremöes and F. Topsøe. Maximum entropy fundamentals. *Entropy*, 3(3):191–226, 2001.

[14] Ahmed E. Hassan and Richard C. Holt. Studying the chaos of code development. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 123, Washington, DC, USA, 2003. IEEE Computer Society.

[15] J. Heaps. *Information Retrieval–Computational and Theoretical Aspects*. Academic Press, Inc., New York, NY, 1978.

[16] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[17] D. Kuck. *The Structure of Computers and Computations, Volume 1*. John Wiley and Sons, New York, NY, 1978.

[18] A. Laemmel and M. Shooman. Statistical (natural) language theory and computer program complexity. Technical Report POLY/EE/E0-76-020, Dept of Electrical Engineering and Electrophysics, Polytechnic Institute of New York, Brooklyn, August 15 1977.

[19] Mario Latendresse and Marc Feeley. Generation of fast interpreters for Huffman compressed bytecode. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 32–40, New York, NY, USA, 2003. ACM Press.

[20] M. Li and P. Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag, New York, 2nd edition, 1997.

[21] M. Li and P. M. B. Vitányi. Kolmogorov complexity and its applications. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity. Elsevier, New York, NY, USA, 1990.

[22] Wentian Li. Bibliography on Zipf's Law, 2005. http://www.nslij-genetics.org/wli/zipf/index.html.

[23] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Jeffrey S. Poulin. *Measuring Software Reuse: Principles, Practices, and Economic Models*. Addison-Wesley, 1997.

[25] David M. W. Powers. Applications and explanations of Zipf's law. In Jill Burstein and Claudia Leacock, editors, *Proceedings of the Joint Conference on New Methods in Language Processing and Computational Language Learning*, pages 151–160. Association for Computational Linguistics, Somerset, New Jersey, 1998.

[26] Jorma Rissanen. *Stochastic Complexity in Statistical Inquiry*, volume 15 of *Series in Computer Science*. World Scientific, 1989.

[27] Sheldon M. Ross. *Stochastic Processes*. John Wiley and Sons; New York, NY, 2nd edition, 1996.

[28] Jeffrey Scott Vitter and Philippe Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 9, pages 431–524. North Holland, 1990.

[29] David Barkley Wortman. *A study of language directed computer design*. PhD thesis, Stanford University, 1973.

A Background

A.1 Asymptotics

Here we recall briefly some facts and notations concerning asymptotic behaviour of functions and series. For a more detailed exposition we suggest [10].

Asymptotic notations. For positive functions $f(n)$ and $g(n)$, we make use of these notations for asymptotic behaviour:

$$f(n) \sim g(n) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1 \tag{13}$$

$$f(n) \prec g(n) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{14}$$

$$f(n) \preceq g(n) \iff \exists c \in \mathbb{R} \;.\; \lim_{n\to\infty} \frac{f(n)}{g(n)} = c \tag{15}$$

The "big-O" style of notation $f \in o(g)$ is equivalent to $f(n) \prec g(n)$. When we write $h(n) \sim g(n) + o(n^2)$ we mean there exists some function $f \in o(n^2)$ such that $h(n) \sim g(n) + f(n)$.

Series and their convergence. A series $\sum_{i=1}^{\infty} a_i$ is convergent when $\lim_{N\to\infty} \sum_{i=1}^{N} a_i$ exists in the standard reals; otherwise it is divergent. The Harmonic series $h_n = \frac{1}{n}$ is divergent, since $\sum_{i=0}^{\infty} h_i = 1 + \frac{1}{2} + \frac{1}{3} + \cdots$ fails to converge.

We shall make use of the following key fact for bounding convergent sequences.

Fact A.1. *Let a_n, b_n be positive sequences. If $\sum_{n=1}^{\infty} a_n$ converges and $\sum_{n=1}^{\infty} b_n$ diverges, then $a_n \prec b_n$.*

Proposition A.1 is useful to establish a bound on the asymptotic growth of a sequence: for example, if $\sum_{n=1}^{\infty} a_n$ must converge, then $a_n \prec \frac{1}{n}$ since the harmonic series diverges.

Advanced Programming Techniques Applied to CGAL's Arrangement Package*

Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin

School of Computer Science Tel-Aviv University, Israel

{wein, efif, baruchzu, danha}@post.tau.ac.il

Abstract

Arrangements of planar curves are fundamental structures in computational geometry. Recently, the arrangement package of CGAL, the Computational Geometry Algorithms Library, has been redesigned and re-implemented exploiting several advanced programming techniques. The resulting software package, which constructs and maintains planar arrangements, is easier to use, to extend, and to adapt to a variety of applications, is more efficient space- and time-wise, and is more robust. The implementation is complete in the sense that it handles degenerate input, and it produces exact results. In this paper we describe how various programming techniques were used to accomplish specific tasks within the context of Computational Geometry in general and Arrangements in particular. A large set of benchmarks assured the successful applications of the adverted programming techniques. The results of a small sample are reported at the end of this article.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Patterns; D.1.5 [**Object-oriented Programming**]

General Terms

Computational geometry, CGAL, arrangements, generic programming, design patterns

1 Introduction

Given a set $\mathcal{C}$ of planar curves, the *arrangement* $\mathcal{A}(\mathcal{C})$ is the subdivision of the plane induced by the curves in $\mathcal{C}$ into maximally connected cells. The cells can be 0-dimensional (*vertices*), 1-dimensional (*edges*), or 2-dimensional (*faces*). The *planar map* of $\mathcal{A}(\mathcal{C})$ is the embedding of the arrangement as a planar graph, such that each arrangement vertex corresponds to a planar point, and each edge corresponds to a planar subcurve of one of the curves in $\mathcal{C}$. Arrangements and planar maps are ubiquitous in computational geometry, and have numerous applications (e.g., [13, 22]), so many potential users in the academia and in the industry may benefit from a generic implementation of a complete software package that constructs and maintains planar arrangements.

CGAL [1], the Computational Geometry Algorithms Library, is the product of a collaborative effort of several sites in Europe and Israel, aiming to provide a generic and robust, yet efficient, implementation of widely used geometric data structures and algorithms. The library consists of a geometric *kernel* [17, 24], which in turn consists of constant-size non-modifiable geometric primitive objects (such as points, line segments, triangles, etc.) and predicates and operations on these objects. On top of the kernel layer, the library consists of a collection of modules, which provide implementations of many fundamental geometric data structures and algorithms. The arrangement package is a part of this layer.

The software described in this paper rigorously adapts, as does CGAL in general, the *generic programming* paradigm [6], making extensive use of C++ class-templates and function-templates. The generic-programming paradigm uses a formal hierarchy of abstract requirements on data types referred to as *concepts*, and a set of components that conform precisely to the specified requirements, referred to as *models*.

In software engineering, *design patterns* are frequently used to supply standard solutions to common problems recurring in software design. Design patterns supply a systematic high-level approach that focuses on the relations between classes and objects, rather than the specification of individual components. See the book by Gamma *et al.* [20] for a catalog of the most common design patterns.

While relations between objects in a design pattern are usually realized in terms of abstract data types and polymorphism, design patterns can successfully be applied in generic programming as well, as we show in this paper. A good example are the point-location algorithms supplied by the arrangement package. One of the most important operations on arrangements is answering the *point-location* query: Given a query point q, find the arrangement cell that contains q. We supply several point-location algorithms, and enable package users to employ the algorithm best suited for their application. To this end, we use the ***strategy*** design-pattern, which defines a family of algorithms, each implemented by a separate class, and we make them interchangeable. The four point-location classes are: `Arr_naive_point_location`, which locates the query point naïvely, by exhaustively scanning all arrangement cells; `Arr_walk_along_a_line_point_location`, which simulates a traversal along an imaginary vertical ray emanating from infinity and directed toward the query point; `Arr_landmarks_point_location`, which uses a set of "landmark" points, whose locations in the arrangement are known. Given a query point, it uses a nearest-neighbor search structure (e.g., KD-tree) to find the nearest landmark and then it traverses the straight

*Work reported in this paper has been supported in part by IST Programme of the EU as a Shared-corst RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes) and by the Hermann Minkowski – Minerva Center for Geometry at Tel Aviv University.

line segment connecting this landmark to the query point. Finally, the `Arr_trapezoidal_ric_point_location` implements Mulmuley's point-location algorithm [29], which is based on the vertical decomposition of the arrangement into pseudo-trapezoids. The last two strategies are more efficient. However, they require preprocessing and consume more space, as they maintain auxiliary data-structures. The first two strategies do not require any extra data and operate directly on their associated arrangements.

In classic object-oriented programming, the point-location process can be realized with an abstract base class that provides a pure virtual function, `locate(q)`, which accepts a point q and results with the arrangement cell containing it. All concrete point-location classes inherit from the base class, and all arrangement algorithms that issue point-location queries use a pointer to an abstract base object, which actually refers to one of the concrete point-location classes. When using generic programming, we rely less on inheritance or virtual functions. Instead, we define a concept named *ArrangementPointLocation_2*, such that all models of this concept must supply a `locate()` function. All the various point-location classes model this concept. Note that the concept definition has no trace in the actual C++ code, so from a syntactical point of view, these classes are completely unrelated. Any generic algorithm that issues point-location queries is implemented as a template parameterized by a point-location class, which is a model of the *ArrangementPointLocation_2* concept.

In the rest of the paper we show how additional design patterns are exploited in the CGAL arrangement package in conjunction with generic programming techniques. The application of combinations of advanced programming techniques is argued to be synergistic. Not only does it make the implementation more generic, it also improves the quality of the software in all measured aspects.

1.1 Related Work

In the classic computational geometry literature two assumptions are usually made to simplify the design and analysis of geometric algorithms: First, inputs are in "general position". That is, degenerate cases (e.g., three curves intersecting at a common point) in the input are precluded. Secondly, operations on real numbers yield accurate results (the "real RAM" model, which also assumes that each basic operation takes constant time). Unfortunately, these assumptions do not hold in practice. Thus, an algorithm implemented from a textbook may yield incorrect results, get into an infinite loop, or just crash, while running on a degenerate, or nearly degenerate, input (see [26, 32] for examples). This is one of the problems addressed successfully by CGAL in general and by the CGAL arrangement package described here in particular.

The need for robust software implementation of computational geometry algorithms has driven many researchers to develop variants of the classic algorithms that are less susceptible to degenerate inputs over the last decade. At the same time, advances in computer algebra enabled the development of efficient software libraries that offer exact arithmetic manipulations on unbounded integers, rational numbers (e.g., GMP — Gnu's multi-precision library [4]) and even algebraic numbers (the CORE [2] library and the numerical facilities of LEDA [5]). These exact *number types* serve as fundamental building-blocks in the robust implementation of many geometric algorithms [37].

Keyser *et al.* [12, 27] implemented an arrangement-construction module for algebraic curves as part of the MAPC and ESOLID libraries. However, their implementations make some general position assumptions. The LEDA library [5, 28] includes geometric facilities that allow the construction and maintenance of arrangements of line segments.

CGAL's arrangement package was the first complete software-implementation, designed for constructing arrangements of arbitrary planar curves and supporting operations and queries on such arrangements. More details on the design and implementation of the previous versions of the package can be found in [18, 23]. Many users (e.g., [11, 14, 21, 25, 31]) have employed the arrangement package to develop a variety of applications.

In this paper we show how concurrent applications of advanced programming techniques improve the quality of the CGAL arrangement software-package, achieving a software design according to the generic-programming paradigm that is more modular and easy to use, and an implementation, which is more extensible, adaptable, and efficient.

1.2 Outline

The rest of this paper is organized as follows: Section 2 provides the required background on CGAL's arrangement package, introducing key terms and presenting its architecture. The four succeeding sections describe the applications of four different design patterns within the generic programming paradigm, namely ***adapter***, ***decorator***, ***observer***, and ***visitor***. These sections detail the pattern intent, their impact, and implementation in the context of the arrangement package. In Section 7 we highlight the performance of our methods on various benchmarks. Finally, concluding remarks and future-research suggestions are given in Section 8.

2 The Architecture

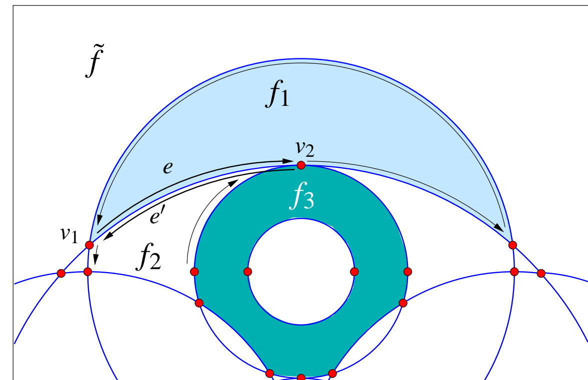


Figure 1. A portion of an arrangement of circles with some of the DCEL records that represent it. $\tilde{f}$ is the unbounded face. The halfedge e (and its twin e') correspond to a circular arc that connects the vertices v_1 and v_2 and separates the face f_1 from f_2. The predecessors and successors of e and e' are also shown. Note that e together with its predecessor and successor halfedges form a closed chain representing the inner boundary of f_1 (lightly shaded). Also note that the face f_3 (darkly shaded) has a more complicated structure, as it contains a hole.

The `Arrangement_2<Traits,Dcel>`[1] class-template represents the planar embedding of a set of (weakly) x-monotone[2] planar curves

[1]CGAL prescribes the suffix _2 for all data structures of planar objects as a convention.

[2]A continuous planar curve C is *weakly x-monotone*, if every vertical line intersects it at most once, or it is a vertical segment.

that are pairwise disjoint in their interiors. It provides the necessary capabilities for maintaining the planar graph, while associating geometric data with the vertices, edges and faces of the graph. The arrangement is represented using a *doubly-connected edge list* (DCEL) — a data structure that enables efficient maintenance of two-dimensional subdivisions.

The DCEL data-structure represents each curve using a pair of directed *halfedges*, one directed from the left endpoint of the curve to its right endpoint, and the other (its *twin* halfedge) going in the opposite direction. The DCEL consists of containers of *vertices* (associated with planar points), *halfedges* and *faces*, where halfedges are used to separate faces and to connect vertices. We store a pointer from each halfedge to the face lying to its left. Moreover, halfedges are connected in circular lists and form chains, such that all edges of a chain are incident to the same face and wind in a counterclockwise direction along its inner boundary (see Figure 1 for an illustration). A non simply-connected face stores a container of *holes*, where each hole is represented by an arbitrary halfedge on the clockwise-oriented chain that forms its outer boundary. The full details concerning the DCEL are omitted here; see [13, Section 2.2] for further details and examples.

The `Arrangement_2` class-template should be instantiated with two objects as follows. (i) A traits class, which provides the geometric functionality, and is tailored to handle a specific family of curves. It encapsulates implementation details, such as the number type used, the coordinate representation, and the geometric or algebraic computation methods. (ii) A DCEL class, which represents the underlying topological data structure, and defaults to `Arr_default_dcel`. Users may extend this default DCEL implementation, as explained in Section 3.1, or even supply their own DCEL class, written from scratch.

The two template parameters enable the separation between the topological and geometric aspects of the planar subdivision. This separation is advantageous as it allows users to employ the package with their own representation of any special family of curves, without having any expertise in computational geometry. They should only be capable of supplying the traits methods, which mainly involves algebraic computations. Indeed, several of the package users are not familiar with computational-geometry techniques and algorithms. The separation is enabled by the modular design and conveniently implemented within the generic-programming paradigm. It is a key aspect of the package, has been forced since its early stages, and heightened by the new design.

The interface of `Arrangement_2` consists of various methods that enable the traversal of the arrangement. For example, the class supplies iterators for its vertices, halfedges and faces. The value types of these iterators are `Vertex_handle`, `Halfedge_handle` and `Face_handle`, respectively. The handle classes themselves supply methods for local traversals. For example, it is possible to visit all halfedges incident to a specific vertex using its `Vertex_handle`, or to iterate over all halfedges along the boundary of a face using its `Face_handle`.

Alongside with the traversal methods, the arrangement class also supports several methods that modify the arrangement, the most important ones being the specialized insertion functions. The functions `insert_at_face_interior(C,f)`, `insert_from_left_vertex(C,u)`. (the symmetric function `insert_from_right_vertex(C,u)`,) and `insert_at_vertices(C,u1,u2)` can be used to create an edge that correspond to an x-monotone curve C whose interior is disjoint from existing edges and vertices, depending on whether the curve endpoints are associated with existing arrangement vertices; see Figure 2 for an illustration of the various cases. Note that these insertion functions hardly involve any geometric operations, if at all. They accept topologically related parameters, and use them to operate directly on the DCEL records, thus saving algebraic operations, which are especially expensive when higher-degree curves are involved. Other modification methods enable users to split an edge into two, to merge two adjacent edges, and to remove an edge from the arrangement.

An important guideline in the design is to decouple the arrangement representation from the various algorithms that operate on it. Thus, non-trivial algorithms that involve geometric operations are implemented as free (global) functions. For example, we offer a free `insert()` function for the *incremental* insertion of general curves[3] computing their *zone* (see Section 6.2), and another free `insert()` function for the *aggregated* insertion of sets of general curves, using a sweep-line algorithm. Another important operation implemented as a free function is the computation of the *overlay* of two arrangements (see [13, Chapter 2] and Section 6.1 below).

2.1 The Traits Class

As mentioned in the previous subsection, the `Arrangement_2` class-template is parameterized with a *traits* class that defines the abstract interface between the arrangement data structure and the geometric primitives they use. The name "traits class" was given by Myers [30] for a concept of a class that should support certain predefined methods, passed as a parameter to another class template. In our case, the geometric traits-class defines the family of curves handled. Moreover, details such as the number type used to represent coordinate values, the type of coordinate system used (i.e., Cartesian or homogeneous), the algebraic methods used, and extraneous data stored with the geometric objects, if present, are all determined by the traits class and encapsulated within it.

The traits-class concept is factored into a hierarchy of refined concepts listed in the next paragraph. The refinement hierarchy is generated according to the identified minimal requirements from the traits imposed by different algorithms that operate on arrangements, thus alleviating the production of traits classes, and increasing the usability of the algorithms.

Every model of the traits-class concept must define two types of objects, namely `X_monotone_curve_2` and `Point_2`. The former represents an x-monotone curve, and the latter is the type of the endpoints of the curves, representing a point in the plane. The basic *ArrangementBasicTraits_2* concept lists the minimal set of predicates on objects of these two types sufficient to enable the operations provided by the `Arrangement_2` class-template itself, and the insertion of x-monotone curves that are also non-intersecting in their interiors. Among these predicate are the *point-status* predicate: given an x-monotone curve C and a point p, determine whether p is above, below, or lies on C; and the *compare-to-right* predicate: given two x-monotone curves C_1, C_2 that share a common left endpoint p, determine the relative position of the two curves immediately to the right of p. The set of predicates defined by the *ArrangementBasicTraits_2* concept is also sufficient for answering point-location queries by various strategies, as detailed in the previous section.[4]

[3] A general curve may not necessarily be x-monotone, can intersect the existing arrangement curves, and its insertion location is unknown *a priori*.

[4] The only exception is the "landmarks" strategy, which requires

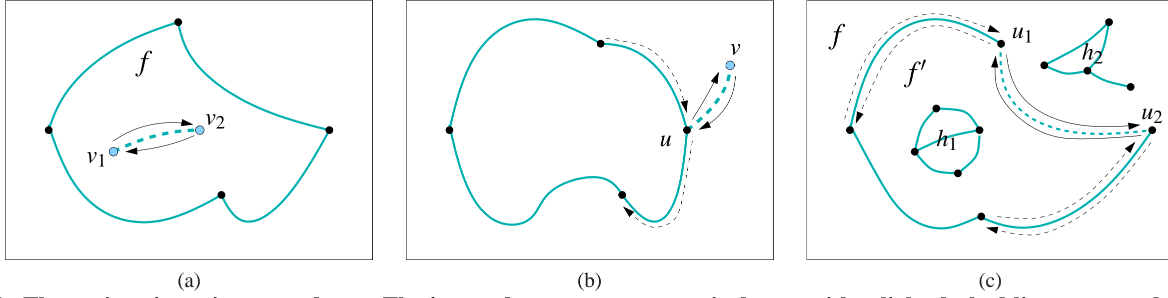


Figure 2. The various insertion procedures. The inserted x-monotone curve is drawn with a light dashed line, surrounded by two solid arrows that represent the pair of twin halfedges added to the DCEL. Existing vertices are shown as black dots while new vertices are shown as light dots. Existing halfedges that are affected by the insertion operations are drawn as dashed arrows. (a) Inserting a subcurve inside the face f. (b) Inserting a subcurve whose one endpoint corresponds to the existing vertex u. (c) Inserting a subcurve whose both endpoints correspond to the existing vertices u_1 and u_2.

The construction of an arrangement of general curves requires the refined *ArrangementTraits_2* concept. In addition to the point and x-monotone curve types, a model of the refined concept must define a third type that represents a general (not necessarily x-monotone) curve in the plane, named `Curve_2`. An intersection point of the curves is of type `Point_2`. In addition, it has to support geometric constructions, such as subdividing a given curve into simple x-monotone subcurves, computing the intersections between two given x-monotone curves, splitting an x-monotone curve into two subcurves at a given point in its interior, and merging two contiguous x-monotone portions of the same curve into a single x-monotone curve.

All traits-class operations are implemented as function objects (*functors*) according to CGAL's guidelines. This allows for the extension of the primitive types above without the need to redefine the methods that operate on them (see [24] for details on the extensible kernel). For a detailed specification of the various concept requirements see [36].

We include several traits classes with the public distribution of CGAL (see Figure 3) as follows. Traits classes for line segments[5], a traits class that operates on continuous piecewise linear curves, namely polylines [23], and a traits class that handles segments of planar algebraic curves of degree 2, namely conic arcs (e.g., ellipses, hyperbolas, or parabolas) [35].

EXACUS [3] is an ongoing project that aims to provide a set of libraries for efficient and exact algorithms for curves and surfaces. In particular, it includes CGAL-compatible traits-classes for computing arrangements of planar algebraic curves of degree 2 (conics) [10], 3 (cubics) [15] and 4 (quartics) [9]. Another traits class for conics was developed as part of an initial attempt to provide a CGAL kernel that supports curved objects [16].

---

a traits class that models the refined *ArrangementLandmarkTraits_2* concept. For lack of space, we omit the details here.

[5]The "non-caching" classes shown in Figure 3, which model the *ArrangementBasicTraits_2* and *ArrangementTraits_2* concepts respectively, directly operate on the kernel segments. Their implementation is simple, yet may lead to a cascaded representation of intersection points with exponentially long bit-length, which in turn may drastically increase the time consumption of arithmetic operations. The class `Arr_segment_traits_2` avoids this cascading problem by storing extra data with each segment. It achieves faster running times when arrangements with relatively many intersection points are constructed. However, it uses more space.

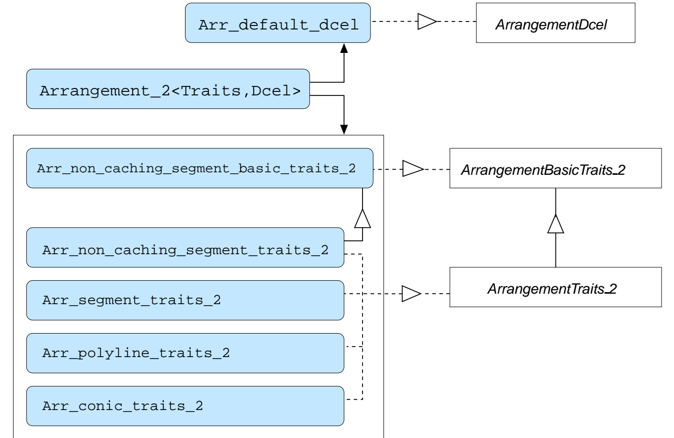


Figure 3. The main `Arrangement_2` class and its template parameters. Arrows designate pointers, solid lines directed through a triangle mark an inheritance or a refinement relation, and directed dotted lines directed through a triangle designate "is a model of" relation.

3 Adapters

The ***adapter*** design-pattern "*converts the interface of a class into another interface clients expect. Adapters let classes work together that could not otherwise, because of incompatible interfaces*" (Gamma *et al.* [20]).

Adapters manifest themselves in a few places in the arrangement module, the first being a mediator between the arrangement class operations and the traits-class primitive operations. This traits adapter add geometric predicates to the traits class, based on the primitive operations provided by a model of the *ArrangementBasicTraits_2* concept. For lack of space we omit the technical details, which can be found in [19].

3.1 The DCEL Face Extender

Another application of an adapter is exhibited in the mechanism to conveniently extend the topological face-feature of the DCEL. While it is possible to store extra (non-geometric) data with the curves or points by extending their types respectively (see more details in Section 4.1), it is also possible to extend the vertex, halfedge,

or face types of the DCEL through inheritance. Many times it is desired to associate extra data just with the arrangement faces. For example, when an arrangement represents the subdivision of a country into regions associated with their population density. In this case, there is no alternative other than to extend the DCEL face. As this technique is might be difficult for inexperienced users, we provide the class-template `Face_extended_dcel<FaceData>`, which extends each face in the `Arr_default_dcel` class with a `FaceData` object.

3.2 Boost Graph Adapters

The BOOST graph library (BGL; see [33]) is a generic library of graph algorithms and data structures designed in the same spirit as STL. It supports graph algorithms, and as our arrangements are embedded as planar graphs, it is only natural to extend the DCEL with the interface that the BGL expects, and gain the ability to perform the operations that the BGL supports, such as shortest-path computation. We adapt an `Arrangement_2` instance to a BOOST graph by providing a set of free functions that operate on the arrangement features and conform with the relevant BGL concepts.

We mention that besides the straightforward adaptation, which associates a vertex with each DCEL vertex and an edge with each DCEL edge, we also offer a *dual* adapter, which associates a graph vertex with each DCEL face, such that two vertices are connected, iff there is a DCEL halfedge that separates the two corresponding faces. These representations are useful for many applications, such as answering motion-planning queries (see e.g., [25]).

4 Decorators

The ***decorator*** design-pattern "*attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality*"(Gamma *et al.* [20]).

In traditional object-oriented programming, attaching additional functionality to an entire hierarchy of classes, all inheriting from a common (perhaps virtual) base class, referred to as the component class, requires the introduction of a decorator class that inherits from the base class and stores a pointer to a virtual component object. When applying one of the methods to the decorator, it first calls the component method, and then performs the supplementary operations. In the arrangement package we apply the decorator design-pattern when we attach auxiliary data to the geometric entities defined by a specific traits class.[6]

4.1 Meta-Traits Classes

We offer several traits-class decorators, which we refer to as *meta-traits* classes. Recall that the traits classes do not have a common base class, but they all model the *ArrangementTraits_2* concept. The meta-traits decorators are parameterized by such a traits class. They inherit some of the base-traits class functors, while overriding others exploiting the auxiliary data maintained with the geometric objects.

The `Arr_consolidated_curve_data_traits_2<BaseTraits,Data>` class inherits its `Curve_2` and `X_monotone_curve_2` types from the respective types of the base-traits class, while extending the curve with an additional *data* field, and the x-monotone curve with a container of data fields. It relies on the geometric operations supplied by the base-traits, and only needs to maintain the extra data fields. When subdividing a curve into x-monotone subcurves, its data field is copied to the resulting subcurves. Similarly, when splitting an x-monotone curve, its data container is duplicated and stored with the two resulting subcurves. When two x-monotone curves overlap, the union of their data containers is computed and stored at the resulting overlapping subcurve.

The `Arr_merged_curve_data_traits_2<BaseTraits,Data,Merge>` class operates similarly, except that it extends the `X_monotone_curve_2` type with just a single data field. When an overlap occurs, it uses the `Merge` functor, given as a template parameter, to merge the data fields of the two overlapping x-monotone curves, and stores the result with the resulting overlapping subcurve.

4.2 Arrangements with History

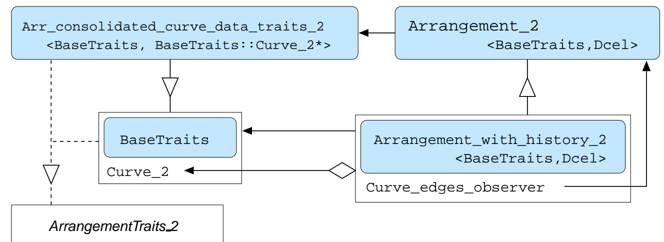


Figure 4. The `Arrangement_with_history_2` decorator. An arrow with a rhombus-shaped tail mean that a class stores a container of objects of the pointed type.

Another major component of the CGAL arrangement package is the `Arrangement_with_history_2<BaseTraits,Dcel>` class-template, which maintains a planar arrangement of general curves, while maintaining its construction history. The input curves that induce the arrangement are split into x-monotone subcurves that are pairwise disjoint in their interior. These subcurves are associated with the arrangement halfedges. In particular, each edge stores a pointer to the input curve associated with it, (or a container of pointers in case the edge is associated with an overlapping section of several curves), while each subcurve stores the set of edges it induces. Users can traverse through the origin curves of each arrangement edge, or iterate on all edges induced by a given input curve.

The `Arrangement_with_history_2` class is not more than a simple decorator for the `Arrangement_2`, as shown in Figure 4. It inherits from an arrangement class that is parameterized by the consolidated curve-data traits (see Section 4.1), where the extra data type is a pointer to a `BaseTraits::Curve_2` object. Thus, the pointers from each edge to its origin curve(s) are automatically maintained. The cross-pointers between input curves and arrangement edges are maintained using an *observer* (see the next section) that keeps track of each change that involves an arrangement edge.

Tracing back the curve (or curves) that induced an arrangement edge is essential in a variety of applications that use arrangements, such as robot motion planning (see e.g., [25]).

5 Observers

The ***observer*** design-pattern "*defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically*" (Gamma *et al.* [20]).

[6]This is a straightforward alternative to extending the DCEL vertices and halfedges (see Section 3.1).

Observers play a significant role in the new design of the arrangement package. They serve many different needs with a single unified approach, as multiple observers can be attached to the same arrangement instance. An important set of observer classes is the one employed by some of the *point-location* strategies that maintain auxiliary data-structures (see Section 1). Another important reason for supporting observers of arrangements is to allow users to introduce their own observer classes. This is not just a convenience, but crucial to the usability of the package, as it might be the only way for providing certain output — data that should be bound with the topological features of the arrangement and is available only during construction. This is explained in Subsection 5.3. In the following subsections we give a detailed description of the notification mechanism implemented via the observer design-pattern.

5.1 The Notification Mechanism

The `Arr_observer<Arrangement>` class-template is parameterized with an arrangement class. It stores a pointer to an arrangement object, and is capable of receiving notifications just before a structural change occurs in the arrangement and immediately after such a change takes place. Hence, each notification is comprised of a pair of "before" and "after" functions. The `Arr_observer<Arrangement>` class-template serves as a base class for other observer classes and defines a set of virtual notification functions, giving them all a default empty implementation. Naturally, one of the objectives is to minimize the observer interface, that is, identifying the minimal set of event points, while capturing all possible changes that arrangements can undergo.

The set of notification functions can be divided into three categories as follows (see [36] for a detailed specification): (i) Notifiers of changes that affect the entire topological structure. Such changes occur when the arrangement is cleared or when it is assigned with the contents of another arrangement. (ii) Notifiers of a local change to the topological structure. Among these changes are the creation of a new vertex, the splitting of an edge, and the formation of a new hole inside a face. (iii) Notifiers of a global change initiated by a free function, and called by the free function (e.g., incremental and aggregate insert; see Section 2). It is required that no point-location queries (or any other queries for that matter) are issued between the calls to the "before" and "after" functions of this pair.[7]

Each arrangement object stores a list of pointers to `Arr_ observer` objects, and whenever one of the structural changes listed in the first two categories above is about to take place, the arrangement object performs a *forward* traversal of this list and invokes the appropriate function of each observer. After the change has taken place the observer list is traversed in a *backward* manner (from tail to head) and the appropriate notification function is invoked for each observer. This allows for the nesting of observer objects. The observer list is not made public, and can only be accessed by the `Arr_observer` class. A free function may choose to trigger a similar notification, which falls under the third category above.

A pointer to a valid arrangement object must be supplied to the constructor of an `Arr_observer` object. The newly created observer object adds itself to the observer list of the arrangement. From that moment on, it starts receiving notifications whenever the associated arrangement object changes. In case the new observer is attached to a non-empty arrangement, its constructor may extract the relevant data from the non-empty arrangement using various traversal methods offered by the public interface of the `Arrangement_2` class, and update any internal data stored in the observer.

[7]This constraint can improve the efficiency of the maintenance of auxiliary data-structures for the relevant point-location strategies, as explained in the next subsection.

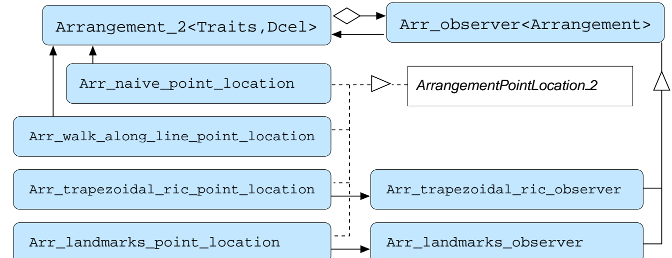


Figure 5. The point-location classes and the notification mechanism.

5.2 Point-Location Observers

As mentioned in Section 1, the *landmarks* and the *trapezoidal* point-location classes maintain auxiliary data structures. These strategies are characterized by very efficient query time but less efficient preprocessing time and space. Naturally, these strategies exhibit better overall performance when the number of updates to the arrangement is relatively small compared to the number of issued queries. Nevertheless, when the arrangement is modified the classes that implement these point-location strategies must keep their auxiliary data structure synchronized with their attached arrangement-instance.

To this end, the landmarks point-location class and the trapezoidal point-location class define the nested observer classes that inherit from `Arr_observer`, and are used to receive notifications whenever the arrangement is modified (see Figure 3). For example, a variant of the landmarks strategy uses the arrangement vertices as landmarks, so whenever a new vertex is created (by the insertion of a new edge or by the splitting of an existing edge), it should be inserted to the nearest-neighbor search structure maintained by the landmarks class. The usage of the notification mechanism makes it possible to associate several point-location objects with the same arrangement simultaneously.

5.3 User-defined Observers

In addition to the point-location observer classes, users can inherit their own observer classes from `Arr_observer` and use the notification mechanism for a variety of purposes, such as dynamically maintaining the extra data they store with the arrangement features. Assume, for example, users associate some additional data records with the arrangement faces (see Section 3.1). In this case their application needs to be notified whenever a new face is created (split from another face) or deleted (merged with another face), and receive a handle to the edge whose insertion (or deletion, respectively) causes this change. An appropriately written observer is ideal for this purpose.

6 Visitors

The ***visitor*** design-pattern "*represents an operation to be performed on an object or on the elements of an object structure. Visitors allow the definition of new operations without changing the classes of the elements on which they operate*" (Gamma *et al.* [20]).

Arrangements have numerous applications, and different applications may require distinct and unrelated operations to be performed on arrangements. Each of these operations may treat different elements of the arrangement data-structure differently using a subset of related operations. Implementing all these operations within the arrangement class and distributing all the operation subsets across the various elements of the arrangement data structure leads to a "polluted" system that is hard to understand, use, and maintain. The BGL, for example, uses visitors [33, Section 12.3] to overcome this problem when extending its graph algorithms.

In the arrangement package we use visitors to implement geometric algorithms that are based on a common algorithmic infrastructure. We have identified two main sets of algorithms: Algorithms based on the sweep-line framework and algorithms based on the zone-computation framework. Thus, we provide two class-templates, namely `Sweep_line_2` and `Arrangement_zone_2`, which implement these two fundamental algorithmic procedures common to the two families of algorithms. Specific algorithms are implemented as visitor classes that receive notifications of the events handled by the basic procedure and can construct their output structures accordingly. The main benefit we gain from this design is a centralized, reusable and easy to maintain code. Moreover, users may add their own sweep-based (or zone-based) algorithms, as the implementation of such an algorithm reduces to implementing an appropriate visitor class.

6.1 The Generic Sweep-Line Algorithm

Sweeping the plane with a line is one of the most fundamental paradigms in Computational Geometry. The famous *sweep-line* algorithm of Bentley and Ottmann [8] was originally formulated for sets of non-vertical line segments, with the "general position" assumptions that no three segments intersect at a common point and no two segments overlap. An imaginary vertical line is swept over the input set from left to right, transforming the static two-dimensional problem into a dynamic one-dimensional one. At each time during the sweep a subset of the input segments intersect this vertical line in a certain order. The order of the segments may change as the line moves along the x-axis, implying a change in the topology of the arrangement, only at a finite number of *event points*, namely intersection points of two segments and left endpoints or right endpoints of segments. The event points, namely segment endpoints and all intersection points that have already been discovered, are stored in a dynamic event queue, named the *X-structure*, in an xy-lexicographic order, while the ordered sequence of segments intersecting the imaginary vertical line is stored in a dynamic structure called the *Y-structure*. Both structures are maintained as balanced binary trees.

The `Sweep_line_2<Traits,Event,Subcurve,Visitor>` class-template implements a generic sweep-line algorithm that can handle any set of arbitrary x-monotone curves [34], containing all possible kinds of degeneracies [13, Section 2.1], [28, Section 10.7], using a small set of geometric predicates and constructions involving the curves. The `Traits` parameter should be instantiated with a model of the *ArrangementTraits_2* concept (see Section 2.1). The `Visitor` parameter should be a model of the *SweepLineVisitor_2* concept, whose functionality is explained in details next.

The `Sweep_line_2` class uses two auxiliary data types: `Event_base`, which stores a `Point_2` object representing the coordinates of an event point, and `Subcurve_base`, associated with a portion of an x-monotone curve (represented as an `X_monotone_curve_2` object) whose interior is disjoint from all other subcurves at the current location of the sweep line (it may intersect undiscovered subcurves as the sweep line advances). These two auxiliary types also store additional data members, needed internally by the sweep-line algorithm, and are not exposed to external users. However, the visitor class may extend these types by inheriting an `Event` class and a `Subcurve` class from the respective base classes and using the extended types to initialize the sweep-line template.

During the sweep-line process the event objects in the X-structure are sorted lexicographically and the subcurve objects are stored in the Y-structure. The `Sweep_line_2` class performs only the very basic operations of maintaining the X-structure and the Y-structure, while the visitor class is responsible for producing the actual output of the algorithm. Whenever the sweep-line class handles an event, it sends a notification to its visitor, with the relevant `Event` object and the `Subcurve` objects incident to it.[8] This way the sweep-line visitor is capable of attaching auxiliary data members and adding functionality to the event and subcurve objects. It can also construct the output accordingly.

It should be mentioned that Bartuschka *et al.* [7] made an initial attempt to provide a generic sweep-line algorithm in the LEDA library. They offer a class that couples a sweep-traits class with a visitor. However, in their implementation the traits class is responsible for performing the entire sweep-line algorithm, whereas our class performs the sweep-line process by itself, and only requires a traits class that supplies a small set of geometric primitives.

A simple sweep-line visitor class is used for reporting all intersection points induced by a set of input curves.[9] This visitor does not require storing any auxiliary data structures with events or with subcurves. The default `Event_base` and `Subcurve_base` types are used to instantiate the sweep-line template. The visitor simply reports an event point p, if it has more than a single incident subcurve.

As mentioned above, a key operation implemented with the aid of a sweep-line visitor is the construction of a DCEL that corresponds to the arrangement of a set of input curves. The visitor class in this case is more complicated, as it needs to store extra data with the subcurves and the events as follows. The event class is extended by a handle of a DCEL vertex that corresponds to the event point. As long as the vertex has not been created yet, the handle is invalid. The subcurve class is extended by a pointer to an event-object point that corresponds to the left endpoint of the subcurve. When processing an event point p, it is possible to go over all subcurves such that p is their right endpoint (so they lie to the left of p) and use this auxiliary data to insert the subcurves into the arrangement using one of the specialized insertion methods (see Section 2). In fact, additional information, stored with each subcurve, helps performing the insertion in the most efficient manner, utilizing all available geometric and topological information. For lack of space, we omit the related technical details here.

Another operation closely related to the construction of a DCEL structure from scratch is the aggregated insertion of new curves into an *existing* arrangement and efficiently updating an existing DCEL structure. In this case we have to sweep over a consolidated set of

[8] The visitor accepts two iterators defining the range of incident subcurves in the Y-structure, so it may also access the neighboring subcurves from above and below.

[9] This operation is indirectly related to arrangements, as it is implemented using the sweep-line framework.

curves comprised of all subcurves associated with existing DCEL edges, and the set of new curves $\mathcal{C}$. Our goal is to discover the intersections involving the new curves and to update the existing DCEL accordingly. We first define a meta-traits class that extends the x-monotone curve type (see [19] for details) with a pointer to a corresponding DCEL halfedge (this pointer will be null for the newly inserted curves).[10] This way we can easily identify events that involve only existing subcurves, which can be ignored, and handle only those events involving the newly inserted curves. When handling such events, we should insert new edge pairs into the DCEL, representing the subcurves of $\mathcal{C}$. In addition, if we locate an intersection between a new curve and an existing subcurve in the DCEL, we should split the corresponding edges at the intersection point to form two halfedge pairs. This operation is elementary and takes constant time.

A fundamental operation that is straightforwardly implemented using a sweep-line visitor is the overlay of two arrangements, given as a "blue" DCEL and a "red" DCEL. The major added difficulty over the previously mentioned visitors is the need to update face structure and face information. Let us assume that each of the input-arrangement faces is associated with some data object (see Section 3.1). If we put our arrangements one on top of the other, we get an arrangement, whose faces correspond to overlapping regions of the blue and red faces. We would like to construct an output DCEL whose faces are associated with the corresponding pairs of blue and red data objects. We do so by sweeping through a consolidated set of "blue" and "red" subcurves. As explained above, it is convenient to use a meta-traits class that extends the x-monotone curves with a color identifier (`BLUE` or `RED` in our case) and a halfedge pointer. This way we can ignore "monochromatic" intersections and compute only the red–blue intersection points (or overlaps). The overlay visitor is parameterized by an overlay-traits class, which defines the merge operations between "red" and "blue" DCEL features.

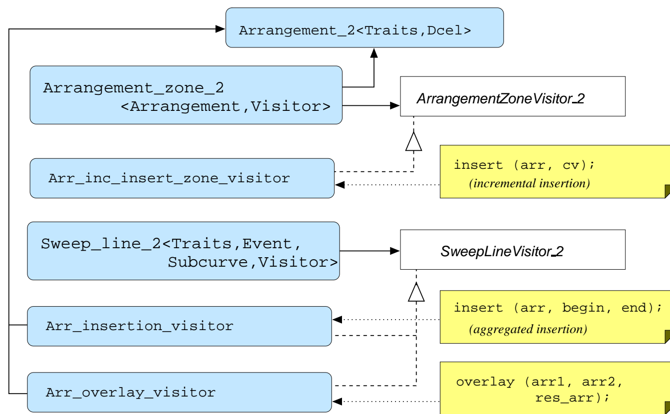


Figure 6. The free functions that are implemented with the aid of visitor classes.

6.2 Zone-Computation Visitors

Many applications can make use of the following operation: Given an arrangement $\mathcal{A}$ and an x-monotone curve C, compute the *zone* of C in $\mathcal{A}$. That is, identify all arrangement cells that the curve crosses. The zone can be computed by locating the left endpoint of C in the arrangement and then "walking" along the curve to the right endpoint, keeping track of the vertices, edges and faces crossed on the way (see for example [13, Section 8.3] for the computation of the zone of a line in an arrangement of lines).

The primary usage for the zone-computation algorithm is the incremental insertion of an x-monotone curve into the arrangement. However, it is sometimes necessary to compute the zone of a curve in an arrangement without actually inserting it. In other cases, the entire zone is not required: Suppose we wish to check whether a given curve passes through an existing arrangement vertex. If such a vertex exists, the process can be terminated as soon as the vertex is located.

While the sweep-line algorithm operates on a set of input x-monotone curves, and its visitors can use the notifications they receive to construct their output structures, the zone-computation algorithm operates on an arrangement object, and its visitors may modify the same arrangement instance as the computation progresses. This makes the interaction of the main class with its visitors slightly more intricate.

The `Arrangement_zone_2<Arrangement,Visitor>` class-template implements a generic zone-computation algorithm. It is parameterized by an arrangement class and by a visitor class. Given a curve C, the zone visitor is notified whenever a maximal subcurve $\hat{C}$ of C is found. The interior of every reported subcurve does not coincide with any arrangement vertex or edge and lies within a face f. The arrangement features that define the subcurve endpoints are also reported. A similar notification is issued whenever a subcurve $\hat{C}$ that overlaps an arrangement edge is detected. In both cases, the visitor returns a pair comprised of a halfedge handle and a Boolean flag as a result. In case the visitor inserts the subcurve $\hat{C}$ into the arrangement, it returns a handle to the newly created edge. Otherwise, it returns an invalid handle. The Boolean value indicates whether the zone-computation process should terminate — this is convenient for gaining efficiency in some applications.

The visitor class `Arr_inc_insert_zone_visitor_2` performs the incremental insertion of an x-monotone curve. It implements the two functions described above to insert the generated subcurves by splitting the halfedges intersected by the curve and using the specialized insertion functions. Other zone visitors are even easier to implement.

7 Experiments

A user of the package has to select the appropriate component in many categories (e.g., number type, geometric kernel, traits class, end point-location strategy). For each selection the user is offered many options. The use of generic programming enables this flexibility. However, it induces a vast number of configurations that must be tested, verified, and tuned. We have developed a benchmarking toolkit that automatically generates all the required configurations and measures the performance of each configuration on a set of inputs. Naturally, we had to restrict ourselves and publish just the most efficient configurations for each traits class. Table 7 indicates the time (in seconds) it took to construct arrangements of various curve types *using exact computations*. For each traits class we have an input file containing many degeneracies (denoted *Degn.*) and a randomly generated input file (denoted *Rand.*). The results, produced by experiments conducted on a Pentium 1.8 GHz, clearly show the major improvement in performance that the package has

[10] It is also possible for the visitor to extend the `Subcurve` type, but if we attach the auxiliary data at the traits-class level we can benefit from giving more efficient implementations of some traits-class functions. For example, we do not have to compute intersections between two existing DCEL subcurves.

undergone from the last public release of CGAL (version 3.1) to the current internal release (version 3.2).

Table 1. Time consumption in seconds of the construction of arrangements of various curve types. The number of input curves and the dimensions of the resulting arrangements are also shown.

Name	C	V	E	F	3.1	3.2
Segments						
Degn.	104	1504	2704	1202	0.170	0.083
Rand.	100	1129	1958	831	0.160	0.041
Polylines						
Degn.	10	112	204	94	0.081	0.020
Rand.	10	1508	2923	1417	0.769	0.223
Conics						
Degn.	41	507	1042	537	2.970	0.647
Rand.	30	677	1303	628	118.0	18.2

The reimplemented package is at least twice as efficient as the old version (CGAL 3.1) in all cases, and as much as six times more efficient in some cases. The main contribution to the improvement is due to the reduction in the number of calls to geometric operations (provided by the traits class). The effect of this reduction increases with the increase in time consumption of the geometric operation. Thus, construction of arrangements of conic arcs exhibits the largest improvement. Figure 7 shows the arrangement of the CGAL logo. It consists of 34 circles and 425 line segments. It took 1.14 seconds to construct the arrangement on the 1.8 GHz Pentium PC using the aggregate insertion method.

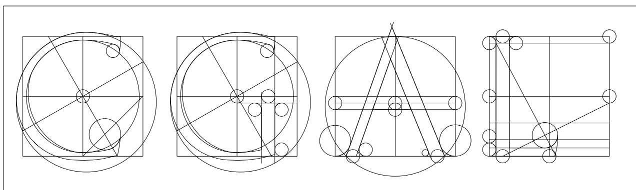


Figure 7. The arrangement of the CGAL logo.

8 Conclusions

We show how our arrangement package can be used with various components and different underlying algorithms that can be plugged in using the appropriate traits classes. Users may select the configuration that is most suitable for their application from the variety offered in CGAL or in its accompanying software libraries, or implement their own traits class. Switching between different traits classes typically involves just a minor change of a few lines of code.

We have shown how careful software design based on the generic-programming paradigm makes it easier to adapt existing traits classes or even to develop new ones. We believe that similar techniques can be employed in other software packages from other disciplines as well.

9 References

[1] The CGAL project homepage. http://www.cgal.org/.

[2] The CORE library homepage. http://www.cs.nyu.edu/exact/core/.

[3] The EXACUS homepage. http://www.mpi-sb.mpg.de/projects/EXACUS/.

[4] The GNU MP bignum library. http://www.swox.com/gmp/.

[5] The LEDA homepage. http://www.algorithmic-solutions.com/enleda.htm.

[6] M. H. Austern. *Generic Programming and the STL.* Addison Wesley, 1999.

[7] U. Bartuschka, S. Näher, and M. Seel. A generic plane sweep framework, 2000. http://www.mpi-inf.mpg.de/ ∼seel/Generic_sweep/index.html.

[8] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, 1979.

[9] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert. EXACUS: Efficient and exact algorithms for curves and surfaces. to appear in proc. 13th Europ. Sympos. Alg. (ESA), 2005.

[10] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In *Proc. 10th Europ. Sympos. Alg. (ESA)*, pages 174–186, 2002.

[11] D. Cohen-Or, S. Lev-Yehudi, A. Karol, and A. Tal. Inner-cover of non-convex shapes. *International Journal on Shape Modeling*, 9(2):223–238, Dec 2003.

[12] T. Culver, J. Keyser, M. Foskey, S. Krishnan, and D. Manocha. ESOLID — a system for exact boundary evaluation. *Computer-Aided Design*, 36, 2003.

[13] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[14] D. A. Duc, N. D. Ha, and L. T. Hang. Proposing a model to store and a method to edit spatial data in topological maps. Technical report, Ho Chi Minh University of Natural Sciences, Ho Chi Minh City, Vietnam, 2001.

[15] A. Eigenwillig, E. S. L. Kettner, and N. Wolpert. Complete, exact and efficient computations with cubic curves. In *Proc. 20th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 409–418, 2004.

[16] I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, and E. P. Tsigaridas. Towards an open curved kernel. In *Proc. 20th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 438–446, 2004.

[17] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. *Software — Practice and Experience*, 30:1167–1202, 2000.

[18] E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The design and implementation of planar maps in CGAL. *The ACM Journal of Experimental Algorithmics*, 5, 2000.

[19] E. Fogel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proc. 12th Europ. Sympos. Alg. (ESA)*, pages 664–676, 2004.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[21] B. Gerkey. Visibility-based pursuit-evasion for searchers with limited field of

view. Presented in the 2nd CGAL User Workshop (2004).

[22] D. Halperin. Arrangements. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. Chapman & Hall/CRC, 2nd edition, 2004.

[23] I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *Proc. 4th Workshop Alg. Eng. (WAE)*, pages 171–182, 2000.

[24] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. 5th Workshop Alg. Eng. (WAE)*, pages 79–90, 2001.

[25] S. Hirsch and D. Halperin. Hybrid motion planning: Coordinating two discs moving among polygonal obstacles in the plane. In J.-D. Boissonnat, J. Burdick, K. Goldberg, and S. Hutchinson, editors, *Algorithmic Foundations of Robotics V*, pages 239–255. Springer, 2003.

[26] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th Europ. Sympos. Alg. (ESA)*, pages 702–713, 2004.

[27] J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: a library for efficient manipulation of algebraic points and curves. In *Proc. 15th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 360–369, 1999.

[28] K. Mehlhorn and S. Näher. LEDA*: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[29] K. Mulmuley. A fast planar partition algorithm, I. *Journal of Symbolic Computation*, 10(3-4):253–280, 1990.

[30] N. Myers. Traits: A new and useful template technique. *C++ Gems*, 17, 1995.

[31] V. Rogol. Maximizing the area of an axially-symmetric polygon inscribed by a simple polygon. Master's thesis, Technion, Haifa, Israel, 2003.

[32] S. Schirra. Robustness and precision issues in geometric computation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1999.

[33] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library, User guide and reference manual*. Addison-Wesley, 2002.

[34] J. Snoeyink and J. Hershberger. Sweeping arrangements of curves. In *Proc. 5th Annu. ACM Sympos. Comput. Geom. (SCG)*, pages 354–363, 1989.

[35] R. Wein. High-level filtering for arrangements of conic arcs. In *Proc. 10th Europ. Sympos. Alg. (ESA)*, pages 884–895, 2002.

[36] R. Wein and E. Fogel. The new design of CGAL's arrangement package. Technical report, Tel-Aviv University, 2005. `http://www.cs.tau.ac.il/ efif/applications/Arr_new_design.pdf`.

[37] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, 2nd edition, 2004.

Reference Counting in Library Design – Optionally and with Union-Find Optimization

Lutz Kettner
Max-Planck Institut Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
kettner@mpi-sb.mpg.de

Abstract

Reference counting has been used and described in abundance. We present novel ideas aimed at class implementations in library design: (1) In library design, generic classes can have variable size, such that an optimal decision for or against reference counting is not possible. We *postpone* this decision to the place of class use. (2) In a context, where equality comparison for the case of equality is expensive, e.g., for exact algebraic number representations, we *unify representations* whenever equality is detected, thus effectively memoizing equality tests. We explain an efficient implementation based on an union-find data structure. (3) Reference counting and *polymorphic class hierarchies* can be combined reusing the pointer in the handle class for the polymorphism. A policy-based generic C++ solution realizes all ideas. *Standard* allocators manage all dynamic memory.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Miscellaneous

General Terms

software library design

1 Introduction

Reference counting has been used and described in abundance. The principle idea is that a dynamically managed resource tracks the number of its users in a reference counter. When the count reaches zero the resource can be released.

In the context of C++, reference counting is frequently presented for smart pointers and string classes; in both cases managing dynamically allocated memory. I am interested in the principal way it is used in string classes. But, let me contrast it to its use in smart pointers first.

Smart pointers are commonly presented as a solution for ownership and resource management problems that plain C/C++ pointers have with dynamically allocated memory. They often preserve the look of reference semantic, at least with (overloaded implementations of) the `operator->()` and `operator*()`. Because of that they offer less opportunities for encapsulation and protection. For example, the copy-on-write strategy for achieving value semantics for mutable, reference-counted objects is not easy to realize.

The use of reference counting in common string classes is motivated by the fact that strings are copied often and are usually sufficiently long, such that reference counting works as an optimization technique. Copying strings becomes extremely efficient for the cost of a small overhead elsewhere. For strings, this cost is particularly negligible since string classes use dynamic memory anyway (otherwise the main efficiency concern for reference counting). Typically, string classes have value semantics. Reference counting is an implementation detail that does not affect the observable interface. We call the class that contains the pointer *handle class*, in distinction to *smart pointer classes*, following the convention from Murray [16], Mehlhorn, and Näher [14, Sect. 13.7].

String implementations offer their functionality in the handle class, e.g., with member functions, while smart pointers refer to the representation class. For example, the length of a string is `s.size()` while the size of some smart-pointed object is `p->size()`. Switching a class to a handle design allows to reuse existing code based on that class while switching to smart pointers requires code changes.

Postpone the Decision

As a library designer, I am particularly interested in the optimization potential of reference counting for classes in a library. However, the effectiveness is not clear for a library developer. The effectiveness depends on the cost for copying (e.g., proportional to the size of the object) versus the cost for dynamic memory allocation plus reference count increments, decrements, and additional space consumption. In addition to these *static costs*, we have the *dynamic* behavior of the program; how often is the object copied compared to the other operations?

For conventional libraries, the static cost for reference counting is often conclusive to decide for or against reference counting: Very small objects are never reference-counted and sufficiently large ones always, because for them the efficiency loss would be a few percent at most. In contrast, for generic libraries with parameterized objects, the static cost is no longer (easily) identifiable in the library itself.

The library designer cannot decide the use of reference counting. Instead, the library user can and should decide. To support this, I present here a solution that a library designer writes an object once, parameterized with a policy template parameter that selects between the two options.

I am interested in geometric algorithms. To implement them robustly yet efficiently, we use exact arithmetic as well as floating-point arithmetic with controlled rounding errors. Flexibility in the coordinate number type is one prime example where a generic

2D point class can be quite large, better using reference counting, or very small, better not using reference counting, respectively. Schirra measures the effect of many such choices in his extensive study [17].

Unify Representations

Exact arithmetic that deals with roots, or algebraic numbers in general, has the interesting asymmetry that detecting equality is much more costly than detecting inequality. This holds for solutions based on separation bounds, such as LEDA reals [14, 18] and CORE [11], as well as for symbolic solutions.

Exact arithmetic number types use reference counting because, based on arbitrary precision integers, they are almost always large. Assume we detect (costly) that two numbers are equal, then we can apply the following optimization: We drop one of the representations and link its handle to the other (by observable behavior identical) representation. The next equality comparison of these two handles detects the identical representation and immediately returns.

Of course, this optimization applies to other objects as well. It is effective if (1) the equality test is expensive enough to justify some additional bookkeeping, (2) sufficiently many equal but not identical objects are tested for equality, and (3) they are tested more than once. It is a quite clever variant of memoization. Compared to caching it has little overhead in bookkeeping and no issues of lifetime or capacity of the cache. Note that it memoizes only equality, not inequality, which is particularly clever for the exact arithmetic example where the memoized result is the expensive one. Complementary, exact representations of algebraic numbers that use isolating intervals would refine these intervals to be disjoint for the case of inequality and then memoize those refined intervals.

We cannot just drop one representation if potentially other handles point to it as well. Reference counting protects us from deleting it, but we would like to have the other handles also profit from the change of representation. The problem can be cast into the well known union-find problem. Essentially, the old representation gets a pointer to the new representation and the handles follow this chain of pointers in the next query. Two optimizations are necessary to make it efficient, explained in detail in Section 3.

We use the policy class mentioned above to select now between the following three options: no reference counting, regular reference counting, and reference counting with the union-find optimization for the equality comparison.

Class Hierarchies and Other Design Options

Reference counting is usually explained for a single representation class, yet, a representation class hierarchy works as well. The pointer in the handle is then a pointer to the base class type and points to derived objects. However, some technical machinery is needed to really combine it elegantly with the other orthogonal choices.

Value semantics is not immediately available with reference counting. One solution is to make objects non-mutable (atomic). Another solution is the copy-on-write strategy, which is supported in our design as well. Lastly, one can choose to adopt reference semantics instead.

For memory allocation, we offer a template parameter for a standard allocator. It will be used for the single representation class as well as for the representation class hierarchy.

We need to answer the question where to put the reference count. The *intrusive* solution places the reference counter in the representation class. Non-intrusive solutions allocate a separate counter, e.g., adding a second pointer to the handle or a forwarding pointer from the counter to the representation class. Flexibility in this question is relevant for generic smart pointers, but we aim at library designers who develop representation classes. So, we choose the intrusive method for its obvious advantages of smaller overhead, which are also documented in the runtime benchmarks reported[1] for the BOOST smart pointer library. Of course, we offer convenient ways to add the reference counter easily.

Paper Outline

The contribution of this paper, besides the novel ideas, is a concrete solution to get all aspects in one coherent design that is easy to use. However, the different aspects interact in non-trivial ways and the final classes are quite intricate. Therefore, I start with a presentation of two examples corresponding to the two main uses of the design; either with a single representation class (*monomorphic use*) or with a class hierarchy (*polymorphic use*). After that, I will present a solution that covers the monomorphic use, and explain the changes necessary to handle the polymorphic use. For completeness, the final solution is in the appendix. Still, several details are omitted here that do not contribute to the understanding of the design and its realization, such as an own namespace, additional constructors, precondition checks, etc., and names have been changed and shortened for the presentation here compared to the implementation that we use in our EXACUS C++ libraries, which are released[2] and can be studied. Before presenting the example uses, I give reference to related work and introduce the union-find data structure and its application here.

2 Related Work

I can trace the origins of our handle-representation implementation back to LEDA[14, Sect. 13.7], the C++ Library of Efficient Data Types and Algorithms. LEDA provides two base classes; `handle_base` for deriving handle types, and `handle_rep` for deriving representations. It is a non-templated solution with overhead implied by a virtual destructor. The solution in LEDA has since then improved to a templated solution.

Koenig and Moo present two plain handle implementations, one intrusive and one non-intrusive [13, Chap. 6,7]. All handle classes look similar. Yet, differences can be seen, for example, in the assignment operator, which shows great similarities between Koenig and Moo's solution, LEDA's and ours, in that it increments one representation before decrementing the other, elegantly avoiding any special case handling of the self-assignment problem.

CGAL, the *Computational Geometry Algorithms Library*[3] [6, 12], uses a templated handle-representation design from the beginning (inspired by LEDA's experience). Initially, all classes in the geometric kernel were reference-counted. Schirra's study [17] confirmed

[1] `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`

[2] `http://www.mpi-inf.mpg.de/EXACUS/`

[3] `http://www.cgal.org/`

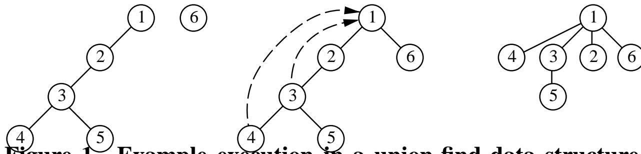


Figure 1. Example execution in a union-find data structure, from left to right: Two sets, one set after `UNION(1,6)`, the same set after `FIND(4)`. The dashed arrows show the relinking caused by `FIND(4)`.

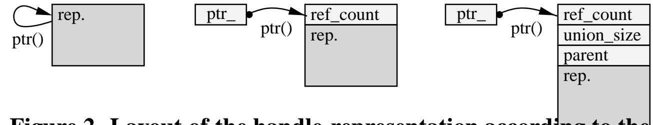


Figure 2. Layout of the handle-representation according to the different policies: (left) In-place layout without explicit pointer stored nor dynamically allocated memory. (middle) Conventional invasive layout of reference counter with representation. (right) Conventional layout enhanced with a counter for the union size and a parent pointer.

that this penalized instantiations with small number types. A parallel series of kernel objects was introduced and later the handle-representation design refined to merge these implementations.

When we started in 2001 the EXACUS project, *Efficient and Exact Algorithms for Curves and Surfaces* [2], I developed the design presented here. Later in the project, the demand for supporting class hierarchies came up. We compute the arrangement of quadrics in space by projecting silhouette curves and pairwise intersection curves into the plane [3]. Silhouette curves are degree two algebraic curves (conics) and intersection curves are (special) degree four algebraic curves. The difference between these curve types is essential for the correctness and efficiency of our solution. The natural design solution is a polymorphic class hierarchy.

C++ programming idioms for handle classes, smart pointers, and reference counting can be found in many C++ text books, ranging from introductions to expert discussions. The older book by Coplien [4] introduces reference-counted handles, intrusive and non-intrusive, as well as a smart pointer for the example of a string class. The older book by Murray [16] contains a description of a string class that uses the handle-representation design, reference counting, and copy-on-write. Horstmann offer an earlier description of smart pointers with reference counting in C++ [9]. Meyers [15, Item 28 & 29] offers a more basic discussion of smart pointers and intrusive and non-intrusive reference counting. Stroustrup discusses a reference-counted string class and a handle-representation design briefly [20, Sect. 15.7]. Josuttis sketches a smart pointer implementation with reference counting to address the question of reference semantics for element storage in standard container types [10, Sect. 6.8]

Vandevoorde and Josuttis describe smart pointers and reference counting [21]. For reference counting, they introduce a policy class for the counter, whether it can be intrusive or not, and whether it needs to protect against concurrent access with threads. Another policy handles deallocation.

Alexandrescu dedicates a chapter in his book [1, Chap. 7] to smart pointers available in his Loki library. He discusses ownership issues, implicit conversion, test and comparison operators, and multi-threading issues. He supports polymorphism and consequential has a policy class that handles the issue of cloning a representation. He mentions copy-on-write, but explains that smart pointers are the wrong place to realize it, since the essential distinction between read-only and write accesses is not possible.

The handle-representation design is an instance of the proxy pattern in the book by Gamma et al. [7], where reference counting and copy-on-write are mentioned as well.

The C++ standard library contains the `auto_ptr` smart pointer template. BOOST's smart pointer library provides intrusive and non-intrusive reference counting.[1] Their proposal is in the process for the next C++ standardization.

3 Union-Find Optimization

A union-find data structure (a.k.a. disjoint sets [5] or partition [14]) maintains a partition for a set K of n keys and a unique representative $\rho(A)$ for each set A in the partition. It provides two operations: (1) For $x \in K$, FIND(x) returns the unique representative $\rho(X)$ for the set X in the partition that contains x. (2) For two sets, X and Y, in the partition, UNION($\rho(X), \rho(Y)$) replaces the two sets with the new set $X \cup Y$ in the partition with a corresponding new unique representative $\rho(X \cup Y)$.

An optimal implementation represents the sets as rooted trees with the keys as nodes and the root node as unique representative. The FIND-operation follows the parent pointers and returns the root node. The UNION-operation links one root node under the other root node. This implementation becomes worst-case optimal if both operations are made a bit smarter: The FIND-operation has cost proportional to the path length to the root node. We perform *path compression*, i.e., each node traversed during a find will be re-linked to point directly to the root node. This does not change the cost of this FIND-operation, but subsequent FIND-operations can be faster. The UNION-operation has constant cost, but it increases the path lengths for the tree linked under the other root node. We use the *linking-by-weight* strategy,[4] in which the tree with fewer nodes is linked below the root node of the other tree. Figure 1 illustrates both operations.

Performing a sequence of UNION-operations and m FIND-operations on a set of size n runs on $O(n + m\alpha(m,n))$ time, where $\alpha(m,n)$ is the slowly growing inverse Ackerman function that is equal to 4 for all practical purposes of the universe. For an elegant proof see Seidel and Sharir [19].

We apply the union-find data structure to the handle-representation scheme with reference counting. Both, the handle and the representation, are nodes in the rooted tree. The pointer in the handle corresponds to a parent pointer, which always points to a representation, i.e., handles never become the representative root node. The representation will be enriched with a parent pointer and a counter for the *union size* defined as the number of all handles and representations that are in the subtree rooted at a representation (including it). The representative root in the tree is the current representation valid for all handles in the tree. Other representations might still exist in the tree because other handles and representations point to them and have not yet been processed in a path compression to link directly to the root node. If they eventually get re-linked, the reference count will drop to zero and the representation will be released

[4] Compared to the *linking-by-rank* strategy, which would be optimal as well and similarly easy to realize.

and removed from the tree.

The memory layout of the different handle-representation choices is illustrated in Figure 2. The difference between in-place and reference counting is abstracted away behind a `ptr()` member function in the handle class that gives access to the representation. Figure 3 shows this data structure in action with a sequence of snapshots on a small example.

The reference count becomes redundant; a representation can be released if its union size drops to one. However, we keep the representation count and its processing in the implementation and presentation, because the union size is an optional feature and readability would suffer.

4 Monomorphic-Use Example

We illustrate our design with a small example of a single representation class that holds an integer value. Representation classes typically are containers for data members and constructors, nothing more. (We ignore access protection.)

```
struct Int_rep {
  int val;
  Int_rep( int i = 0) : val(i) {}
};
```

The common functionality for handle classes is factored into the `Handle` base class template. Its full signature is:

```
template <
  typename T,
  typename Policy = Handle_without_union,
  typename Alloc  = std::allocator<T> >
class Handle;
```

`T` is the representation type that the handle manages. In the monomorphic case, `T` does not contain a reference counter yet. The counter is added automatically by the `Handle` class template. `Policy` is the policy class that determines whether we use reference counting at all and, if so, if we also use the union-find optimization. The default value selects reference counting without the union-find optimization. `Alloc` is a standard allocator with suitable default.

We derive our handle class from `Handle<Int_rep>`. We add a constructor that calls the base class constructor, which in turn calls the representation class constructor at the point of allocation. This avoids unnecessary construction of temporaries on the way. It is realized with template constructors, which have the limitation to be only available for up to a fixed number of arguments, in our own library currently for up to 10 arguments. There exists another way of initializing representations that I skip in this presentation.

```
struct Int_handle : public Handle<Int_rep> {
  typedef Handle<Int_rep>  Base;
  Int_handle( int i = 0) : Base(i) {}
  ...
```

We implement two member functions: A simple access function illustrates the access to the stored value through the (protected) `ptr()` member function in the base class, which internally performs the FIND-operation if required. So, each data access triggers path compression. A corresponding set member function illustrates the copy-on-write strategy.

```
  int  value() const { return ptr()->val; }
```

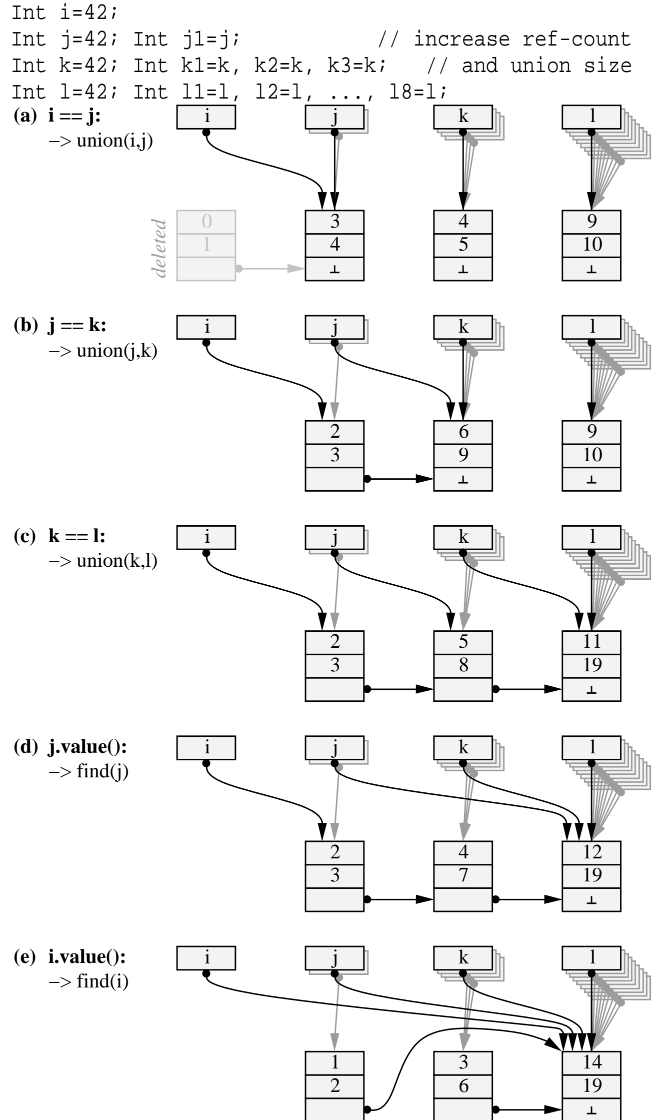


Figure 3. Example of a detailed trace of the union-find data structure at work. We assume a class `Int` similar to the example in Section 4. Following the layout on the right of Figure 1, the representation stores a reference count, a union size, and a parent pointer to another representation.

```
  void set_value( int i) {
    copy_on_write();
    ptr()->val = i;
  }
  ...
```

We conclude with the equality operator and its use of the union-find optimization. The `unify` member function in the base class is called for an argument `i` if the representation of `i` has been found to be equivalent to the own representation. Depending on the policy template argument, this call will do nothing, or it will lead to the simplification that one representation will be replaced by the other as described in the previous section. Note that the UNION-operation determines from the union sizes which representation will be replaced by which. This is of no concern in many applications, but if it does, one could look at the union sizes beforehand and manually preserve the preferred representation. Although reference-counted, the `const&` in the parameter passing is recommended, because otherwise parts of the union-find optimization gain would be delayed to the next FIND-operation on the calling object (since the union would work on a copy of the handle). A note on constness: I designed the classes for the use in a value semantic context where the union-find optimization is an implementation decision not affecting the interface. As such, it is allowed to work on its internal data also for constant objects.

```
  bool operator==( const Int_handle& i) const {
    bool equal = (value() == i.value());
    if (equal)
      unify(i);
    return equal;
  }
};
```

So far, our handle class has a fixed reference counting policy. A template parameter for the policy (and analogously for the allocator) makes it more flexible:[5]

```
template < typename Policy = Handle_without_union>
struct Int_handle : public Handle< Int_rep, Policy>
```

With such an implementation, our handle class has value semantics. The default implementations of copy construction and assignment perform correctly. The difficulties of dynamic memory allocation are encapsulated in the `Handle` class template.

In CGAL, *Geometric Kernel* classes and many *Basic Library* classes, such as Nef polyhedra [8], use a similar handle-representation implementation with value semantics. *Geometric Kernel* objects in the `CGAL::Cartesian` and `CGAL::Homogeneous` kernels are reference-counted and in the `CGAL::Simple_cartesian` and `CGAL::Simple_homogeneous` kernels are non-reference-counted. Kernel objects are immutable and do not use copy-on-write. The Nef polyhedra use reference counting with copy-on-write. The unification strategy is not available.

In the EXACUS libraries, many classes use precisely this handle-representation implementation with reference counting and value semantics.

[5] Note that now calls to member functions of the `Handle` template base class need a `this->` prefix, a consequence of the *two-phase name lookup* rule for templates [21, Sect. 9.4].

5 Polymorphic-Use Example

We define a small class hierarchy of two representation classes, again just holding an integer value. In the previous monomorphic example, we were able to let the `Handle` class template define generically the type that contains the representation class together with the intrusive reference count. Here, the `Handle` class template does not know the class hierarchy and thus cannot provide this convenience. Instead, we require now that the root of the class hierarchy is derived from a base class that provides the reference count. This is not a restriction in our context; the design presented here is not a generic smart pointer that users apply to already existing classes, rather it is a technique for a library developer to apply to newly developed classes.

The base class containing the reference count actually depends on the `Policy` of the `Handle` class template. Furtheron, it needs the allocator as template argument, which will be explained later in detail. This leads to the following base class definition for our root in our part of the class hierarchy. Besides the base class, the difference to the monomorphic example is the required conventional `clone()` member function:

```
template <typename Policy, typename Alloc>
struct Int_rep
: public Policy::Hierarchy_base<Alloc>::Type {
  int val;
  Int_rep( int i = 0) : val(i) {}
  virtual Ref_counted_hierarchy<Alloc>* clone() {
    return new Int_rep( *this);
  }
  virtual int  get_val() const { return val; }
  virtual void set_val( int i) { val = i; }
};
template <typename Policy, typename Alloc>
struct Int_rep2 : public Int_rep< Policy, Alloc> {
  int val2;
  Int_rep2( int i)
      : Int_rep<Policy,Alloc>(i), val2(0) {}
  virtual Ref_counted_hierarchy<Alloc>* clone() {
    return new Int_rep2( *this);
  }
  virtual int  get_val() const { return val2; }
  virtual void set_val( int i) { val2 = i; }
};
```

We turn to the handle implementation. In the monomorphic use, we pass constructor arguments to the `Handle` class template, which in turn uses `Alloc` to allocate and construct the representation. This does not work here. We have to be able to create any of the derived representation classes, which is best done in the handle constructor with the `new` operator. Nonetheless, we want to use the allocator for memory management, which is the reason why we pass it to the base class of the representation class hierarchy. This base class redefines the `new` and `delete` operators to use the allocator. (The allocator of the `Handle` class template is not used in this setting.) We implement two constructors in the handle class as a simple means to select between the two possible representation classes.

```
template < typename Policy,
           typename Alloc = std::allocator<char> >
struct Int_handle
  : public Handle< Int_rep<Policy,Alloc>, Policy>
{
  typedef Int_handle<Policy,Alloc> Self;
```

```
  typedef Handle< Int_rep<Policy,Alloc>, Policy>
    Base;
  Int_handle( int i = 0)
    : Base( new Int_rep<Policy,Alloc>(i)) {}
  Int_handle( int i, int j)
    : Base( new Int_rep2<Policy,Alloc>(i+j)) {}

  int  value() const { return ptr()->get_val(); }
  void set_value( int i) {
    copy_on_write();
    ptr()->set_val(i);
  }
  bool operator==( const Self& i) const {
    bool equal = (value() == i.value());
    if (equal)
      unify(i);
    return equal;
  }
};
```

One may ask, if the differences between the monomorphic use and the polymorphic use are so big, why do we use the same `Handle` class template? We still reuse plenty of the orthogonal aspects in the `Handle` class template, which is illustrated by the fact that the member functions in the handle class are identical in both examples. In particular, copy-on-write and the unification work the same. Of course, one choice is not possible in the polymorphic case, namely to not use dynamically allocated representations (`Handle_in_place` policy), because we would lose the internal pointer used for the realization of polymorphism in the C++ language. Not explained in this paper, but all such constraints are checked statically at compile time.

The affine transformation classes in the CGAL *Geometric Kernel* use a similar handle-representation implementation with a polymorphic class hierarchy of specialized representations for translation, scaling, and others [6]. In EXACUS, the arrangement of quadrics in space uses this solution for the polymorphic representation of projected silhouette curves and pairwise intersection curves [3].

6 Handle Class Template for Monomorphic Use

We offer three policy classes for the `Policy` parameter.

- `Handle_in_place`: the handle stores the representation directly *in place* without reference counting and without dynamic memory allocation. It cannot be used together with a hierarchy of polymorphic representation classes, since the necessary pointer is now missing in the handle.
- `Handle_without_union`: regular reference counting, i.e., without the union-find optimization.
- `Handle_with_union`: reference counting with the union-find optimization.

The first policy is actually not used as a policy class in the generic `Handle` class template. It is used to select a specialization, which mostly consists of empty stub implementations. The specialization is listed in Appendix A.2. The other policies deal with the union-find optimization (two static member functions) and two dependent types. The dependent types are actually member templates and similar to the rebind mechanism of the standard allocator interface.

- The type expression `Policy::Rep_bind<T,is_ch>::Rep` shall define the representation type including the intrusive reference counter. The Boolean `is_ch` is false if the type `T` is not derived from the base class required for the polymorphic class hierarchy and true otherwise. If `is_ch` is true, `Rep` shall be equal to `T`, otherwise it shall be equal to `Ref_counted<T>` if no union-find optimization is used and equal to `Ref_counted_uf<T>` if the union-find optimization is used.
- The type expression `Policy::Hierarchy_base<Alloc>::Type` shall be the base class suitable for deriving a polymorphic class hierarchy, see the example above.
- The static member function `unify(h,g)` acting on two handles `h` and `g` shall perform the union step, if applicable for this policy.
- The static member function `find(h)` acting on one handle `h` shall return a pointer to the currently valid representation. It shall perform the find step with the necessary side effects on all involved representations.

The `Ref_counted` class template combines the intrusive reference counter with the representation type `T`. This is the actual representation a handle refers to. The template with its few supportive member functions is self-explanatory.

```
template <typename T>
class Ref_counted {
  mutable unsigned int count;  // reference counter
  T                    rep;    // representation
public:
  typedef Ref_counted<T> Self;
  typedef T* Rep_pointer;

  Ref_counted() : count(1) {}
  Ref_counted( const T& t) : count(1), rep(t) {}
  Ref_counted( const Self& r):count(1),rep(r.rep){}
  Rep_pointer  base_ptr()  { return &rep; }
  void add_reference()     { ++count; }
  void remove_reference()  { --count; }
  bool is_shared() const   { return count > 1; }
  int  union_size() const  { return 1+count; }
  void add_union_size(int) {}
};
```

Analogously, the `Ref_counted_uf` class template combines the representation type `T` with a reference counter, a counter for the current union size, and a parent pointer. The representation is only valid for roots in the union-find data structure, in which case the parent pointer is null.

```
template <typename T>
class Ref_counted_uf {
public:
  typedef Ref_counted_uf<T> Self;
  typedef T* Rep_pointer;
  friend class Handle_with_union;
private:
  mutable unsigned int count;  // reference counter
  mutable Self*        parent; // parent or 0
  mutable int          u_size; // union set size
  mutable T            rep;    // representation
public:
  Ref_counted_uf() : count(1),parent(0),u_size(2){}
  Ref_counted_uf( const T& t)
      : count(1), parent(0), u_size(2), rep(t)   {}
```

```
  Ref_counted_uf( const Self& r)
      : count(1), parent(0), u_size(2),rep(r.rep){}
  ... // identical functions as in Ref_counted
  bool is_forwarding() const { return parent != 0;}
  int  union_size() const    { return u_size; }
  void add_union_size(int a) { u_size += a;  }
};
```

We begin with a preliminary implementation of the `Handle` class template neglecting polymorphic class hierarchies. We explain the necessary changes for the polymorphic use in the next section. The complete solution is in Appendix A. We have already explained the template signature in Section 4. It follows a type declaration section with the noteworthy `Rep` type for the representation and the allocator that is rebound to the `Rep` type, which might differ from `T`. The `Rep_pointer` pointer type is in the monomorphic use always `T*`. The return type of `find` is `Rep_pointer`.

```
template <typename T,
          typename Policy = Handle_without_union,
          typename Alloc  = std::allocator<T> >
class Handle {
public:
  typedef Handle< T, Policy, Alloc>           Self;
  typedef Policy                      Handle_policy;
  typedef Alloc                           Allocator;
  typedef typename Policy::template Rep_bind< T,
      false>                                   Bind;
  typedef typename Bind::Rep                    Rep;
  typedef typename Rep::Rep_pointer    Rep_pointer;
  typedef typename Alloc::template
    rebind<Rep>::other                    Rep_alloc;
  friend class Handle_without_union;    // policies
  friend class Handle_with_union;
  ...
```

It follows the only data member, a pointer to the representation, as well as a static allocator variable, object allocation, and object release (if the reference count drops to zero).

```
private:
  mutable Rep*      ptr_;
  static  Rep_alloc allocator;
  static  Rep* new_rep( const Rep& rep) {
    Rep* p = allocator.allocate(1);
    allocator.construct(p, rep);
    return p;
  }
  void remove_reference() {
    Policy::find( *this);  // ptr_ is now valid rep
    if ( ! is_shared()) {
      allocator.destroy( ptr_);
      allocator.deallocate( ptr_, 1);
    } else {
      ptr_->remove_reference();
      ptr_->add_union_size( -1);
    }
  }
  ...
```

Next is the protected interface for the derived handle class. It supports the access to the representation through the `ptr()` member function, which always invokes the FIND-operation. Also the other member functions, `unify` and `copy_on_write`, have been used in the examples above.

```
protected:
  T* ptr() {
    return static_cast<T*>(Policy::find(*this));
  }
  const T* ptr() const {
    return static_cast<const T*>(
        Policy::find( *this));
  }
  void unify( const Self& h) const {
    Policy::unify( *this, h); // forward to policy
  }
  void copy_on_write() {
    Policy::find( *this);  // ptr_ is now valid rep
    if ( is_shared() ) {
      Rep* tmp_ptr = new_rep( * ptr_);
      ptr_->remove_reference();
      ptr_->add_union_size( -1);
      ptr_ = tmp_ptr;
    }
  }
  ...
```

The public interface implements the important four parts for resource management classes: constructors, copy-constructor, destructor, and assignment operator.

```
public:
  Handle() : ptr_( new_rep( Rep())) {}
  Handle(const Self& h) {
    Policy::find( h);      // ptr_ is now valid rep
    ptr_ = h.ptr_;
    ptr_->add_reference();
    ptr_->add_union_size( 1);
  }

  template <class T1> explicit Handle( const T1& t)
    : ptr_( new_rep( Rep( T(t)))) {}
  ...            // more constructor templates here

  ~Handle() { remove_reference(); }
  Self& operator=( const Self& h) {
    Policy::find( h);      // ptr_ is now valid rep
    h.ptr_->add_reference();
    h.ptr_->add_union_size( 1);
    remove_reference();
    ptr_ = h.ptr_;
    return *this;
  }
};
```

7 Extension for Polymorphic Use

A subtle difference lies in the `Rep_pointer` definition. It does not point to `T`, which would be the user base class, but to our base class provided for the user, the `Ref_counted_hierarchy` class template. It provides the reference count and the re-defined `new` and `delete` operators to use the allocator template parameter.

```
template <typename Alloc = std::allocator<char> >
class Ref_counted_hierarchy {
  mutable unsigned int count;  // reference counter
  static Alloc alloc;
public:
  void* operator new(size_t bytes) {
    return alloc.allocate( bytes);
```

```
  }
  void  operator delete(void* p, size_t bytes) {
    alloc.deallocate((char*)p, bytes);
  }
  typedef Ref_counted_hierarchy<Alloc> Self;
  typedef Self*                  Rep_pointer;

  Ref_counted_hierarchy() : count(1) {}
  Ref_counted_hierarchy( const Self&) : count(1) {}
  virtual ~Ref_counted_hierarchy() {}

  // Return a copy of myself: Write in all classes:
  // return new Derived_type( *this);
  virtual Self* clone() = 0;
  Rep_pointer base_ptr()   { return this; }
  void add_reference()     { ++count; }
  void remove_reference()  { --count; }
  bool is_shared() const   { return count > 1; }
  int  union_size() const  { return 1+count; }
  void add_union_size(int) {}
};
```

The base class template extends for the union-find optimization to `Ref_counted_hierarchy_uf` to contain the union size and the parent pointer in the obvious way, analogously to `Ref_counted_uf` in the previous section.

The main difference to the monomorphic case is a new protected constructor accepting pointers to newly allocated representations. Many other differences are in the details of allocating and deallocating the representations. The design uses template meta-programming to detect, if the template argument provided for the parameter `T` is derived from our classes. Then we work in the polymorphic-use version or otherwise we work in the monomorphic-use version. In particular, we test for a derivation relationship among two classes and use helpers, such as `Type_from_int<i>` described by Alexandrescu [1], to select among overloads of a function.

8 Union-Find Policy Classes

Defined as an empty struct, the `Handle_in_place` class is not really a policy class since its purpose is to select a specialization of the `Handle` class template. For the other policy classes we omit how the dependent types are selected with template meta-programming techniques, and refer to Section 6 for their specification. What remains are the `unify` and `find` member function templates. They are both trivial for the `Handle_without_union` policy class:

```
struct Handle_without_union {
  template <typename H>
  static void unify( const H& h, const H& g) {}
  template <typename H>
  static typename H::Rep_pointer find( const H& h){
    return h.ptr_->base_ptr();
  }
};
```

The `Handle_with_union` policy class has an additional asymmetric `unify` member function whose first argument is the larger set compared to the second argument. All three member function templates have handles as arguments and switch to the representations `hrep` and `grep`, respectively. They implement the algorithms explained in Section 3:

```
struct Handle_with_union {
  template <typename H>
  static void unify_ls( const H& h, const H& g) {
    // |H| >= |G|, let g point to h's rep
    typename H::Rep* hrep = h.ptr_;
    typename H::Rep* grep = g.ptr_;
    grep->add_union_size(-1);
    if ( grep->is_shared()) {       // grep remains
      grep->remove_reference();
      hrep->add_reference();
      hrep->add_union_size( grep->union_size());
      grep->parent = hrep;
    } else {
      g.delete_rep( grep);              // grep goes
    }
    // redirect handle g and incr. hrep's counter
    g.ptr_ = hrep;
    hrep->add_reference();
    hrep->add_union_size(1);
  }
  template <typename H>
  static void unify( const H& h, const H& g) {
    if ( find(h) !=  find(g)) { // safety check
      if ( h.ptr_->union_size()
         > g.ptr_->union_size())
        unify_ls( h, g); // make g point to h's rep
      else
        unify_ls( g, h); // make h point to g's rep
    }
  }
  template <typename H>
  static typename H::Rep_pointer find( const H& h){
    typedef typename H::Rep Rep;
    if ( h.ptr_->is_forwarding()) {
      Rep* new_rep = h.ptr_; // find new valid rep
      while ( new_rep->parent != 0)
        new_rep = static_cast<Rep*>(
                    new_rep->parent);
      Rep* rep = h.ptr_;
      while ( rep != new_rep) { // path compression
        Rep* tmp = static_cast<Rep*>(rep->parent);
        if ( rep->is_shared()) {     // rep remains
          rep->remove_reference();
          if ( tmp != new_rep) {
            // re-link rep to the new_rep
            rep->parent = new_rep;
            new_rep->add_reference();
          }
        } else {                        // rep goes
          h.delete_rep( rep);
        }
        rep = tmp;
      }
      h.ptr_ = new_rep; // hook h to new_rep
      new_rep->add_reference();
    }
    return h.ptr_->base_ptr();
  }
};
```

9 Discussion

The presented design of a handle-representation implementation in C++ features the novel ideas of a smart union-find optimization and

postpones the decision for reference counting to the library user. The implementation works seamlessly together with standard allocators and polymorphic class hierarchies. My emphasis was on the ease of use and readability of handle classes and representation classes realized with this design (c.f. Section 4 and 5). Monomorphic and polymorphic use can not be exchanged easily, but that is not to be expected unless the monomorphic use loses some of its current ease of use.

The presented implementation was developed for the EXACUS libraries, representing 120 thousand lines of C++ library and test code, and is since 2001 successfully in use for many classes. A similar implementation is in use in the CGAL project for quite some time.

The implementation is exception safe, but does not support multiple threads, which could be added with straightforward locking techniques. Another option, which I have not tried, could be garbage collection instead of reference counting.

10 Acknowledgments

I thank Kurt Mehlhorn for the union-find suggestion and Sylvain Pion for his enthusiastic and resourceful work on the CGAL kernel. I thank Eva Kluge for her support. I am grateful to the anonymous reviewers for their numerous valuable and knowledgeable comments.

11 References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[2] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert. EXACUS: Efficient and exact algorithms for curves and surfaces. In *Proc. 13th Annu. Europ. Symp. Algo. (ESA)*, Oct. 2005. (to app).

[3] E. Berberich, M. Hemmer, L. Kettner, E. Schömer, and N. Wolpert. An exact, complete and efficient implementation for computing planar maps of quadric intersection curves. In *Proc. 21th Annu. Sympos. Comput. Geom.*, pages 99–106, 2005.

[4] J. O. Coplien. *Advanced C++ Programming Styles and Ideoms*. Addison-Wesley, 1992.

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

[6] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. *Software – Practice and Experience*, 30:1167–1202, 2000.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissidis. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] P. Hachenberger and L. Kettner. Boolean operations on 3D selective Nef complexes: Optimized implementation and experiments. In *Proc. of 2005 ACM Symposium on Solid and Physical Modeling (SPM'05)*, pages 163–174, Cambridge, MA, June 2005.

[9] C. S. Horstmann. Memory management and smart pointers. *C++ Report*, March, April 1993. Reprint in *More C++ Gems*, Ed. R. C. Martin, SIGS Reference Library, Cambridge University Press, pp. 33–50, 2000.

[10] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 1999.

[11] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th Annu. Sympos. Comput. Geom.*, pages 351–359, 1999.

[12] L. Kettner and S. Näher. Two computational geometry libraries: LEDA and CGAL. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discr. and Comput. Geom.*, pages 1435–1463. CRC Press LLC, Boca Raton, FL, 2nd edition, 2004.

[13] A. Koenig and B. E. Moo. *Ruminations on C++: A Decade of Programming Insight and Experience*. Addison-Wesley, 1996.

[14] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[15] S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

[16] R. B. Murray. *C++ Strategies and Tactics*. Addison-Wesley, 1993.

[17] S. Schirra. A case study on the cost of geometric computing. In *Proc. of ALENEX'99*, 1999.

[18] S. Schmitt. The diamond operator – implementation of exact real algebraic numbers. In *Proc. 8th Internat. Workshop on Computer Algebra in Scient. Comput. (CASC 2005)*, LNCS 3718, pages 355–366. Springer, 2005.

[19] R. Seidel and M. Sharir. Top-down analysis of path compression. *SIAM Journal of Computing*, 34(3):515–525, 2005.

[20] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[21] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003.

APPENDIX A: IMPLEMENTATION

The full implementation of the `Handle` class template supports polymorphism and has a specialization for the `Handle_in_place` policy. We skip the series of obvious forwarding template constructors.

A.1 Handle Class Template

```
template <typename T,
          typename Policy = Handle_without_union,
          typename Alloc  = std::allocator<T> >
class Handle {
public:
  typedef Handle< T, Policy, Alloc>           Self;
  typedef Policy                      Handle_policy;
  typedef Alloc                           Allocator;
  typedef Conversion_derived_base<T,
    Ref_counted_hierarchy_base>               Check;
  enum { is_ch = Check::is_inheritance };
  typedef Type_from_int< Check::is_inheritance> CH;
  typedef Type_from_int<false>              T_false;
  typedef Type_from_int<true>                T_true;
  typedef typename Policy::template Rep_bind< T,
      is_ch>                                   Bind;
```

```
  // internal representation, i.e., T plus a ref
  // count (if needed) or just T if we derive
  // from the base class to support a class
  // hierarchy for the representations.
  typedef typename Bind::Rep                   Rep;
  typedef typename Rep::Rep_pointer    Rep_pointer;
  typedef typename Alloc::template
    rebind<Rep>::other                   Rep_alloc;
  friend class Handle_without_union;
  friend class Handle_with_union;

private:
  mutable Rep*      ptr_;
  static  Rep_alloc allocator;

  static Rep* new_rep( const Rep& rep) {
    Rep* p = allocator.allocate(1);
    allocator.construct(p, rep);
    return p;
  }
  static void del_rep( Rep* p, T_false) {
    allocator.destroy( p);
    allocator.deallocate( p, 1);
  }
  static void del_rep( Rep* p, T_true) { delete p;}
  static void del_rep( Rep* p) { del_rep(p, CH());}

  static Rep* clone_rep( Rep* p, T_false) {
    return new_rep( *p);
  }
  static Rep* clone_rep( Rep* p, T_true) {
    return static_cast<Rep*>(p->clone());
  }
  static Rep* clone_rep( Rep* p) {
    return clone_rep( p, CH());
  }
  void remove_reference() {
    Policy::find( *this);  // ptr_ is now valid rep
    if ( ! is_shared()) {
      del_rep( ptr_);
    } else {
      ptr_->remove_reference();
      ptr_->add_union_size( -1);
    }
  }
  template <class TT>
  Rep* make_from_single_arg( const TT& t, T_false){
    return new_rep( Rep( T(t)));
  }
  template <class TT>
  Rep* make_from_single_arg( TT t, T_true) {
    return t; // has to be ptr convertible to Rep*
  }
protected:
  T*        ptr()       {
    return static_cast<T*>(Policy::find(*this));
  }
  const T* ptr() const {
    return static_cast<const T*>(
        Policy::find( *this));
  }
  void unify( const Self& h) const {
    Policy::unify( *this, h); // forward to policy
  }
  void copy_on_write() {
    Policy::find( *this);  // ptr_ is now valid rep
    if ( is_shared() ) {
      Rep* tmp_ptr = clone_rep( ptr_);
      ptr_->remove_reference();
      ptr_->add_union_size( -1);
      ptr_ = tmp_ptr;
    }
  }
  Handle( Rep* p) : ptr_( p) {} // for hierarchies
public:
  Handle() : ptr_( new_rep( Rep())) {}
  Handle(const Self& h) {
      Policy::find( h);    // ptr_ is now valid rep
      ptr_ = h.ptr_;
      ptr_->add_reference();
      ptr_->add_union_size( 1);
  }
  // Forwarding constructor passing its parameter
  // to the representation constructor. In case of
  // the class hierarchy of representation classes,
  // this constructor is also chosen for pointers
  // to newly allocated representations that are
  // types derived from T. In that case, the ptr
  // is just assigned to the internal pointer.
  template <class T1>
  explicit Handle( const T1& t)
      : ptr_( make_from_single_arg( t, CH())) {}
  ...             // more constructor templates here
  ~Handle() { remove_reference(); }
  Self& operator=( const Self& h) {
      Policy::find( h);    // ptr_ is now valid rep
      h.ptr_->add_reference();
      h.ptr_->add_union_size( 1);
      remove_reference();
      ptr_ = h.ptr_;
      return *this;
  }
};
```

A.2 Handle Class Template Specialization

```
template <typename T, typename Alloc>
class Handle<T, Handle_in_place, Alloc> {
public:
  typedef Handle< T, Handle_in_place, Alloc>  Self;
  typedef Handle_in_place            Handle_policy;
  typedef Alloc                          Allocator;
  // identify T with the internal repr. Rep.
  typedef T                                    Rep;
private:
  Rep  rep;    // store the rep in place
protected:
  T*       ptr()       { return &rep; }   // access
  const T* ptr() const { return &rep; }
  void     unify( const Self&) const {}      // NOP
  void     copy_on_write()           {}      // NOP
public:
  Handle() {}                        // constructors
  Handle(const Self& h) : rep( h.rep) {}
  template <class T1>
  explicit Handle( const T1& t) : rep( Rep(t)) {}
  ...             // more template constructors here
};
```

A Rationale for Semantically Enhanced Library Languages

Bjarne Stroustrup
Department of Computer Science
Texas A&M University
College station, TX-77843
and AT&T Labs – Research
bs@cs.tamu.edu

Abstract

This paper presents the rationale for a novel approach to providing expressive, teachable, maintainable, and cost-effective special-purpose languages: A *Semantically Enhanced Library Language* (a *SEL language* or a *SELL*) is a dialect created by supersetting a language using a library and then subsetting the result using a tool that "understands" the syntax and semantics of both the underlying language and the library. The resulting language can be about as expressive as a special-purpose language and provide as good semantic guarantees as a special-purpose language. However, a SELL can rely on the tool chain and user community of a major general-purpose programming language. The examples of SELLs presented here (*Safe C++*, *Parallel C++*, and *Real-time C++*) are based on C++ and the Pivot program analysis and transformation infrastructure. As part of the rationale, the paper discusses practical problems with various popular approaches to providing special-purpose features, such as compiler options and preprocessors.

1 Introduction

We often need specialized languages. Researchers need to experiment with new language features, such as concurrency features [24], facilities for integration with databases [5], and graphics [4] . Developers can sometimes gain a couple of orders of magnitude reductions in source code size with corresponding reductions in development time and defect rates, by using such special-purpose languages in their intended domains. Unfortunately, such special-purpose languages are typically hard to design, tedious to implement, expensive to maintain, and — despite their obvious utility — tend to die young.

Using a (special-purpose) library is an obvious alternative to a special-purpose language. However, a library cannot express or exploit semantic guarantees beyond what its host language provides. The basic idea of *Semantically Enhanced Library Languages* (SEL Languages or simply SELLs) is that when augmented by a library, a general-purpose language can be about as expressive as a special-purpose language, and by subsetting that extended language, a tool can provide about as good semantic guarantees. Such guarantees can be used to provide better code, better representations, and more sophisticated transformations than would be possible for the full base language. For example, we can provide support for parallel operations on containers as a library. We can then analyze the program to ensure that no undesirable access to elements of those containers occurs — a task that could be simplified by enforcing a ban of languages features that happened to be undesirable in this context. Finally we can perform high-level transformations (such as parallelizing) by taking advantage of the known semantics of the libraries.

Like a library, a SELL can benefit from the extensive educational, tools, and library infrastructure of the base language. Therefore, the cost of designing, implementing, and using a SELL is minuscule compared with a special-purpose language with a small user base. Examples will be based on ISO standard C++ supported by the Pivot infrastructure for program analysis and transformation (5.2). The focus will be on templates because they provide the key mechanism for statically type-safe expression of advanced ideas in C++.

What is called a "special-purpose language" here is often called a domain-specific language (e.g. [10]). Distinctions can be made between the two terms, but none that appear relevant to the discussion here, so please consider those two terms as equivalent in this context.

The organization of this paper is

1. Introduction
2. State some ideals for support of software development and maintenance.
3. Present some of the — usually fatal — problems that face new programming languages.
4. Discuss a few alternative approaches, such as dialects and macro languages.
5. Focus on the SELL approach and the way it can be supported in C++ using the Pivot.
6. Sketch the design of a few SELLs: type-safe C++, Parallel C++, and Real-time C++.
7. Conclusions

2 Ideals

For every specific problem area, we can design a special-purpose language that exactly matches the desired syntax and semantics of the domain and the desires of the programmers that will use that language. In an ideal world, no general- purpose language can match such a special-purpose language when applied in its specific problem area. When a special-purpose language has been done perfectly, there is a one-to-one correspondence between the fundamental concepts of the application domain and the language constructs. Given that, the language constructs can be minimal and directly reflect the terminology of the field as found in common use and major textbooks.

This is not a new ideal. Fortran did a good job at that task for arithmetic in the 1950s and COBOL successfully attacked the business processing needs of the time. Since then, thousands of languages have been designed for specific domains and almost as many have been designed to try to be able to effectively express that ideal for less specific domain. Lisp and Simula originated the two main approaches to more directly express application domain concepts directly in code: the functional and object-oriented approaches. In these languages, and in their numerous offspring, a set of concepts is represented as a library of related functions or classes. In such general-purpose and near-general-purpose languages the ideal of the perfect language for the task takes the form of libraries.

What do we expect from a well-designed special-purpose language? Concise notation is the beginning. Consider a simple, common, and useful example:

```
A = k*B + C
```

First note the algebraic notation using operators. Notation is important for concise expression of key ideas in a community. This particular notation is based on almost 400 years of history in the mathematics/scientific community.

Essentially all languages can handle `A=k*B+C` when the variables denote scalar values, such as integers and floating point numbers. For vectors and matrices, things get more difficult for a general-purpose language (that doesn't have built-in vector and matrix types) because people who write that kind of code expect performance that can be achieved only if we do not introduce temporary variables for `k*B` and `k*B+C`. We probably also need loop fusion (that is, doing the element *, +, and = operations in a minimal number of loops). When the matrices and vectors are sparse or we want to take advantage of known properties of the vectors (e.g., `B` is upper-triangular), the library code needed to make such code work pushes modern general purpose language to their limit [17, 23] or beyond — most mainstream languages can't efficiently handle that last example. Move further and require the computation of `A=k*B+C` for large vectors and matrices to be computed in parallel on hundreds of processors. Now, even an advanced library requires the support of a non-trivial run-time support system [1]. We can go further still and take advantage of semantic properties of operations, such as "remembering" that C was the result of an operation that leaves all its elements identical. Then, we can use much simpler add operation that doesn't involve reading all the elements of `C`. For other examples, preceding the numerical calculation with a symbolic evaluation phase, say doing a symbolic differentiation, can lead to immense improvements in accuracy and performance. Here, we leave the domain where libraries have been considered useful. Reasoning like that and examples like that (and many more realistic ones) have led to the creation of a host of special-purpose languages for various forms of scientific calculation [24].

So, the ideal notation offered by a general purpose language is just the beginning. It can be the basis for comprehension, for fast compilation, for performance (exploiting type information and semantic properties), for reasoning about programs (by the implementation or associated tools), for programmer productivity, for making facilities accessible to professionals who need to program in their field of expertise, yet don't want to become professional programmers (e.g., physicists, engineers, animators, and graphical designers). Finally, the clarity of the code can greatly ease maintenance.

Note that the ideals and strengths of special-purpose and general-purpose languages can conflict. By definition, a general-purpose language aims at allowing the programmer to express just about anything. On the other hand, a special-purpose language gains much of its strength from allowing a programmer to express only what makes sense in its specific domain. When it comes to program analysis and optimization, this is a great strength of a special- purpose language. For example, if an optimizer tries to do a symbolic differentiation of a program in a language focused exclusively on scientific computation, it does not have to worry about a programmer trying to differentiate the draw function of a graphics system.

Convenient graphical interfaces are often associated with special-purpose languages. They can be used as an extreme example of direct representation of ideas or as a special- purpose language. However, such interfaces can be used to equal effect for code in a general-purpose language, so GUIs will not be examined further here.

3 Problems

It is fun to design a new programming language. Doing the initial implementation and trying the new language with clever examples can be most exhilarating. However, it is plain hard work to bring the implementation up to the level needed for users who care nothing about language design subtleties. Building supporting tools, such as debuggers and profilers, is hard work and not intellectually stimulating for most people who design programming languages. Real users also need basic numeric libraries, basic graphical facilities, libraries for interfacing with code written in other languages, "hand holding" tutorials, detailed manuals, etc. Doing each of those things once can be interesting and most educational, doing them all or repeatedly is tedious and often expensive. Porting the implementation, tool base, and key applications to new machines, platforms, and compilers repeatedly is not only tedious, but also career death for many people. Basically, designing, implementing, maintaining and supporting a language is tremendously expensive. Only a large user community can shoulder the long-term parts of that.

The net effect is that on the order of 200 new languages are developed each year and that about 200 languages become unsupported each year. "Language death" doesn't just happen to bad languages. For example, you can find a collection of 16 languages for high-performance computing in *Parallel programming using C++* [24]. Most have very appealing aspects, many are based on brilliant insights, all were supported by an enthusiastic research group, and all had years of stable funding. None are in major use today. None are supported by an organization outside the one that developed them. All but one are dead.[1] Interestingly, the one survivor (Charm++) is more of a library than a language.

In addition to the really ambitious language design projects, thousands of researchers work on dialects and associated tools for their research. Such dialects are not built from scratch; instead, a compiler and key support tools are modified to serve the new dialect. Essentially all become unsupported upon graduation, funding expiration, tenure, promotion, transfer of maintenance responsibilities, change of fashion, change of any part of the tool chain, change of management, consolidation of IT operations, etc.

Some of these languages are designed for research only (or claim to be), but many are aimed at non-research use (or claim to be) and

[1] I'd love to be proven wrong on this, so if you have a counter example, please tell me and we'll celebrate this exceptional success together.

most language designers harbor dreams of wide use for their languages. However, most of these new languages and dialects never see non-research use. The ones that do, are generally unloved by maintenance organizations. That is not just prejudice and unwillingness to learn or to change. There are perfectly good reasons for the lack of enthusiasm in maintenance organizations. For example, the supply of reasonably priced support personnel tends to be severely limited. Good designers and good researchers (typically with PhDs) rarely want to become maintainers with a typical maintainer's salary, work conditions, and career prospects.

Each new language and dialect has its own tool chain that needs to be kept current and in sync with other tools. The cost of doing so for a minor dialect is typically higher than for a major language — because the cost of the latter is amortized over millions of users. These reasons are often solid in economic and management terms, even though they can be heartbreaking for the proponents of a new language or dialect. For example, the largest application using ML within AT&T was rewritten in a non-research language and so were the few uses of a very interesting rule-based language R++ that can be seen as an early precursor of aspect-oriented programming [11].

Tool chain problems don't just happen to "Mom&Pop languages." I have seen major organizations abandon Ada for just this reason. Similarly, education can be a major problem. If a language isn't taught in universities (or only in a few schools), good programmers become scarce and most organizations cannot afford to re-train new hires. Furthermore, new programmers are sometimes overly impressed by their favorite language and resist training. I have seen organizations abandon Fortran for that reason. The two effects are mutually reinforcing.

However, most special-purpose languages, proprietary dialects, etc., never get a large enough user base and tool set to worry about decline. Most minor and research languages simply never gain the tool support and availability on a wide range of platforms that users of mainstream languages take for granted. Unless a new language is really a minor dialect of an existing language, almost all of the design and implementation effort is recreating facilities — such as debuggers, profilers, database interfaces, and GUI interfaces — that tend to lie outside the main interest of the language designers. This repetitive reconstruction of "standard facilities" provided for other languages breeds lots of "good little ideas" as people add improvements. Unfortunately, such "little improvements" tend to further isolate users. Since "further isolates" can be read as "locks-in users" as well as "provides better support than the competition," there is often little resistance to gratuitous replication and incompatibility. Compatibility is just hard work, and typically unrewarding.

How many users does it take to sustain an infrastructure? Of course, that depends on a lot of things, but generally it requires more people than work on a single application. In fact, it typically takes at least a small company. That is more — often significantly more — people than it took to create the initial design and implementation of a language. If — as is usual — these people have to be paid from the revenues from sales and teaching, a special-purpose language now comes under pressure to become more widely useful. That is, the special-purpose language starts to offer facilities for general computation, general data structures, access to "external systems," database facilities, graphics facilities, etc. The result can be summarized as "Every special-purpose programming language wants to grow up and become a general-purpose programming language." Typically, this is a precursor to "language death" (because of instability, lack of design focus, and added cost) or to a retreat into a commercially viable niche that covers only a small part of the special-purpose language's natural application domain. This withdrawal is often accompanied with a lot of commercial hype and a tendency to hide and obscure genuine technical information.

Many (probably most) special-purpose languages suffer from "edge effect" problems. The "edge effect" (also more evocatively known as the "falling off the cliff" effect) comes when a programmer needs to do something that isn't supported by the special-purpose language. For example, a programmer using a language for specifying interactive graphics might want to say "when viewed from a sufficient distance, groups of objects may be considered one object." The graphics system could have provided such a feature, but in this case it didn't (and the difference in real-time response was about a factor of 100). What does the programmer do? By definition, every special-purpose language has such "edges." For students and novices, the effect can be a nuisance; for professionals working on large projects (such as the airline control application from which this graphics example was chosen), the result can be the abandonment of the special-purpose language in favor of an alternative, such as a graphics library written in a general-purpose language. But what does a programmer do if changing tools isn't an option? In a "pure" special-purpose language, a new primitive operation or object must be added. That's not something every application programmer can do because it may effect the basic model of the special-purpose language. I have seen the time for adding a simple feature vary from one day (ask a local expert and wait for the overnight tool build) to half a year (wait for the next release) or more. This kind of delay can kill a project, so it must be considered among the risks when choosing or designing tools. For a library — and for any tool that allows a programmer to add code written in a general-purpose language — the problem is minor.

The final nail in the coffin of many special-purpose languages is that once it is designed and in use, it is relatively easy to "emulate" its facilities in a general-purpose language. Often, the value of a special-purpose language is not really in the language implementation or its particular syntax (though programmers can be passionately devoted to a syntax). The value is in the design, the programming model, the techniques for use, and possibly some special algorithms or data structures sustaining applications. Typically, those special-purpose language "implementation details" can be separated from the language and used directly from a general-purpose language. This is all the easier because these key components are written in some general-purpose language. All that is needed for their direct use is a nice programming interface in that general-purpose language. The definition of "nice" will reflects the experience gained from the use of the special purpose language.

Please note that a language is rarely "killed" by any one of the problems mentioned. Typically, the language succumb to a combination of problems. Also, this list is not intended to be complete or necessary "fatal": some special-purpose languages do survive and some fail because of reasons not listed here. An exhaustive list of problems probably couldn't be compiled, and if it could it would be beyond the scope of this paper.

3.1 Case study: R++

A detailed study of a few hundred new languages to provide solid evidence for the observations made here would be useful. However, I doubt it would dampen the enthusiasm for designing new languages. Here, I'll just present one small example, and then proceed to an alternative approach to providing new facilities for programmers.

R++[11] is an unrecognized precursor to aspect-oriented programming. Basically, it is an extension of C++ in which you can define actions and triggers for actions. For example, a retirement policy can be associated with an `Employee` class like this:

```
rule Employee::retirement_policy {
     age>=65 && status!=retired
=>
     cout << name << " must retire...";
}
```

This is simple enough to be easy to teach. Furthermore, the implementation was a small enough increment on C++ that it was relatively easy to maintain. Since R++ is a superset of the general-purpose language C++ there are no edge effects. It was used in a reasonably large telecom operations system application. Tutorials, academic papers, manuals, experience reports, implementation, etc. were provided. You can find them on the web [11].

For all practical purposes, it died in 1996. The reasons were basically that the porting and training costs were too high compared to the benefits. What do I mean by dead? Completely unused? Not necessarily. Ever so often, I see a reference to R++ and I'd be surprised if there wasn't a project somewhere using it. Probably, there are also a couple of research groups trying it out. However, despite ideas that appear fundamentally sound, despite avoiding edge effects by being embedded in a general-purpose language, and despite having an implementation that did sustain a major application, R++ still suffered many of the various problems mentioned in this section and failed to gain major use outside its originating organization.

4 Alternatives

So, in most cases, designing a new language is not an economically viable solution to the problem of how to provide special-purpose facilities. A language often looks good for a few years but maintenance, porting, education, etc. is too expensive and the result is death or at best stagnation of the tool chain and the user community. As a technical/economical choice, designing a new language most often is a mistake. Most language design efforts soak up resources reinventing a few wheels and then die having provided a poor return on investment. The resources could have been better spent on improvements to an existing major language and its libraries and tools. Furthermore, most new languages divide a community by creating barriers to communication of new ideas and not infrequently by generating hype that trigger language wars and distrust of new ideas. Not all of the problems are the fault of the new language, and new ideas must be explored and exploited. So, what else can we do to bring the ideal of direct expression of ideas in code into wider use?

So, let's assume that we are in one of the many situations where designing a new language is likely to be uneconomical and to have undesirable effects on the spread of ideas. What alternatives do we have when our task is to provide programmers with improved tools for expression ideas in code?

Here are some popular approaches:

1. Compiler options and pragmas
2. Libraries
3. Preprocessed languages
4. Dialects

Each can be an effective approach in some cases and each has been used in ways that have been deemed successful. Here, we must consider their fundamental and practical strengths and weaknesses.

These are not the only possible approaches. For example, one might consider:

1. Dynamically typed languages
2. A new, more general, general-purpose language

Dynamically typed languages are not considered here. The the main reason is an interest in compile-time guarantees. Basically, dynamically -typed languages constitutes a different world from the statically typed world that I focus on here. Dealing with that world is beyond the scope of this paper.

One might consider building a new general purpose language providing facilities that are so complete that every special-purpose language can be expressed directly through the mechanism of the general purpose language. That's one of the holy grails of general-purpose language design. In fact, over the last 30 years or so, there has been a stream of such languages offering facilities for defining extended syntax (e.g., through embedded parsers) and associating semantics with the newly defined constructs. Such languages are also beyond the scope of this paper. Part of the reason is that providing such a language is beyond the means of most organizations needing a special-purpose language. Another problem is that (ironically) such languages themselves suffer from the problems of being special-purpose languages with small user communities and insufficient support. The success rate for general-purpose languages is even lower than the rate for special-purpose languages.

4.1 Compiler options and pragmas

People who add compiler options and/or pragmas rarely think that as language design. In particular, (in the C and C++ worlds) a `#pragma` can be ignored by a compiler. However, every new `#pragma` and compiler option introduces a new dialect. It is something to consider when building a system, when specifying a system configuration, when porting a system, when documenting a system, and when trying to understand application code. Assume for a moment that options and `#pragmas` are not used for back-door language extension. Then, they are simply insufficient for doing anything really interesting in the direction of better expression of ideas. Most special-purpose languages require additions. Also, they often require restriction of use of certain undesirable language features. That makes compiler options a too crude a mechanism. Options tend to apply indiscriminately; for example, we might want to eliminate the use of `goto`. However, the option will then eliminate all `gotos` — even the acceptable ones for breaking out of loops in a highly optimized matrix implementation and the essential ones in implementation of the state machines generated from a high-level modeling library/language. What is needed is to distinguish between uses of an undesirable language feature in user code and their use in the implementation of trusted components. Compiler options are best left for conventional uses, such as backwards compatibility switches; `#pragmas` are best avoided.

4.2 Libraries

Libraries can provide expressive power and notational convenience that approximate that of built-in language features. However, it is

hard to ensure consistent use of a library (or a set of libraries). It is even harder to ensure consistent use of a subset of a library when — as is common — too much has been bundled into a single unit of distribution. Other language features can interfere with what a library attempts to achieve. The C++ standard library is a classical example. It provides well-behaved containers, but some programmers use arrays instead and thereby prevent any meaningful guarantees to be made for the program as a whole.

When ambitious in what they try to achieve in terms of generality or performance, libraries can become very elaborate and brittle. For example, some C++ template meta-programming libraries aiming at very general support for high-performance numerical computation reach their goal at the cost of complete obscurity of implementation details that becomes visible to users during debugging. Often, a library breaks the zero-overhead principle in search for generality.

A library cannot, by itself, eliminate basic problems with host language semantics. For example, in C and C++, aliasing problems persists so that a library cannot provide guarantees needed for confidence, transformations, and optimizations. Often, a library is (at least partially) defined in terms of its implementation; it is not specified as an entity separate from its host language implementation. This is not a fundamental problem, but it is a common problem, and often a serious one in comparison to a special-purpose language.

4.3 Preprocessed languages

Generating code from a higher-level language into a lower-level one has been popular for decades. For example early C compilers generated assembly code; early C++ compilers generated C code; GUI builders, CAD systems, IDL processors, modeling languages, etc., generate code in languages such as C, C++, Java, C#. That is, the language source is preprocessed into a host language. The resulting languages and language processors are referred to by many names, such as preprocessors, macros, generators, wizards, builders, and meta-languages. One way of distinguishing an implementation of a language implemented by such techniques from a facility defined by such translation techniques is whether you can ever get an error message from the target language compiler. If you can it's a preprocessor; if not it's a compiler. For example, by that criteria, the original C and C++ translators (into assembler and C) were compilers whereas Ratfor, C macros, and Microsoft "wizards" rely on preprocessors. C++ templates are "right on the edge" in that they receive some compiler support (and will receive significantly more in the future: concepts [21, 20]). However, compiler error messages sometimes fail to refer to the original template source and often do so spectacularly badly. In consequence, some programmers consider templates "like macros" and avoid them; many more avoid uses they consider nontrivial. Here, we consider preprocessed languages, rather than abstraction facilities integrated within a language.

The language (generator, macro-language, modeling language, whatever) defined by a preprocessor becomes yet another special-purpose language. It requires documentation, training, tool support. In particular, you need to use a preprocessor together with a matching tool chain and compiler. Unless the preprocessor is integrated into the tool chain and shipped with every implementation, this implies lock-in and slow upgrades. It is not uncommon for the preprocessor not to work with the most current version of the compilers and tools or the underlying language. The main reason is that the preprocessor implementer doesn't get access to those compilers significantly before their own users. This commonly leads to users having to make a painful choice between using the preprocessor or the latest and greatest compiler and other tools. This creates friction between the preprocessor users and any non-preprocessor users they collaborate with. The debugging, compatibility, and portability problems persist because old compilers don't just die. It can take a large organization the better part of a decade to get everyone upgraded to the latest version (of something), just to fall behind again at the next release. For example, it took "forever" (almost a decade) to get C++ template implementations good enough for mainstream use. However, some users still rely on decade old compilers.

A preprocessed language tends to have problems interacting with the type system of the host language. Having the same type system as the host language is often not good enough — after all, the purpose of a preprocessed language is to elegantly express things that cannot be expressed elegantly in the host language. Error detection and error reporting problems are just the most obvious examples of this. Concepts (a type system for types) [21, 18], as being developed to improve C++ templates' support for generic programming and template metaprogramming, is an example of a mechanism addressing the problem of mismatch of the type systems of a higher-level language and a lower-level host language. Higher order types fills some of the same role in the specification of abstractions in functional languages.

So, a preprocessed language share many problems of with a special-purpose language with a stand-alone implementation. In fact, as their tools become more complete and their definition more precise and separate from the host language, they grow into special-purpose languages. Conversely, if their implementation and type system support becomes more integrated with the host language, they cease to be separate languages and become abstraction mechanisms of the host language (C++ templates is a prominent example). In addition, preprocessing languages tend to suffer the problems of libraries: Unless all code conforms to the conventions of the preprocessed language, the guarantees the language can rely on and offer weaken.

4.4 Dialects

Take a popular general-purpose language, add desired features to a compiler and/or a run-time support system, and you have your own private dialect. This may be the most popular way of creating a new language. The result is not quite a special-purpose language, but it has special-purpose features embedded in a general-purpose language. Working in a production-quality general-purpose language implementation is hard, though. Many people will simultaneously be making modifications in such an implementation. Furthermore, compilers, debuggers, libraries, tools are required parts of such implementations and major implementations target many platforms. Consequently, most people who extend a language in this way do so in a minor — less messy — implementation, modifying only the part of the tool chain they need, and target only the platforms they care about. This is reasonable — in many cases even essential — to allow people to focus their efforts on the design and implementation of the new facilities they want. Unfortunately, the effect is that unless the major vendors adopt the new dialect, its designers are left with a private language. This implies all the usual private language costs — and the usual mortality rate. In addition, it is essentially impossible to remove undesirable features from a dialect. Doing so would destroy compatibility and basically move the language away from the dialect classification and into the special-purpose language classification.

5 The SELL approach

The analysis in sections 3 and 4 paints a grim picture of the problems of applying language design and implementation techniques to support software development. One conclusion would be to leave the field to big corporations with deep pockets: Let them do the design, development, and apply their marketing muscle; then we live with the results, whatever they may be. An alternative conclusion is to withdraw into some cosy ghetto of our own design and let the rest of the world do what it likes without interference or input from us. I like neither alternative and point to a way to dodge the horns of this dilemma:

1. superset: Add libraries to provide application-specific facilities, then
2. subset: Subtract features (outside the library implementation) to provide semantic guarantees

The result is a subset of a superset of a language called a *Semantically Enhanced Library Language*. When subsetting we can aim at a "clean and regular" language. Since a SELL will aim for a narrower application domain than its host languages, we have a good chance of the result being simpler than its host.

We must consider this approach in terms of expressiveness ("can we really express things as well in a library as in a special-purpose language?") and tools ("will we get stuck developing and maintaining a messy tool chain?"). The claim is that the answers can be "yes" and "no" for a large enough range of problems and a low enough cost to prefer the SELL approach over the traditional approaches mentioned in sections 3 and 4. Obviously, the SELL approach is not completely new — in fact, it is an attempt to synthesize what has worked best in the traditional approaches and dodge the worst problems. Please also note that I don't claim that the other approaches to making special-purpose features available never work or that there are no other alternatives. That would be absurd. What I do claim is that the success rate for new languages — if measured by survival of a language for a decade and use outside the group that originated it — is very low and the costs higher than often realized.

The argument about expressiveness of libraries is based on a pair of old Bell Labs sayings:

1. Library design is language design
2. Language design is library design

We need both. In other words, the expressiveness of a library depends on the ability of a general-purpose language to define libraries. Functional programming, object-oriented programming, and generic programming are prominent schools of thought that give a prominent role to library building.

The skills needed to write a good library are very similar to the skills needed for all high-end systems programming or application building. Furthermore, when we write a library, we can rely on existing infrastructure (compilers, debuggers, libraries, education, etc.). The result is that libraries are cheap to produce compared to alternatives.

However, the tools part could easily lead us into the debugging, tool chain, and maintenance problems characteristic of dialects and preprocessors. To avoid that we need a tool for expressing constraints and high-level transformations that is minimally invasive into the tool chain. To further keep the tool problems under control, we need a general tool for doing that and one that will fit into all tool chains. That is, we need a general-purpose tool for analyzing source code and performing source-level transformations that relies on a standard interface to compilers.

5.1 C++

In principle, any general-purpose programming language can be the host language for the SELL approach. Unsurprisingly, my favorite/chosen host language is C++ [19, 8]

C++ has the virtues of stretching to a very broad range of application areas, good performance, a large and lively user community, and support for compilers, libraries, and tools for essentially all platforms [22].

C++'s abstraction facilities provide adequate support for object-oriented programming, generic programming, traditional procedural programming, and multi-paradigm programming combining elements of those. Classes plus templates plus overloading is the basis of expressiveness and performance.

Obviously improvements are possible — even given the Draconian compatibility constraints imposed by the huge user community and the wide range of application areas. In particular, we hope that the next standard (C++0x) will offer concepts (a type system for types), more general and flexible facilities for initialization, and remedies for many minor annoyances [20]. Unfortunately, the compatibility constraints and the use of C++ for very low-level system components precludes remedying obvious weaknesses, such as overly aggressive implicit conversions (inclunding the array-to-pointer conversion) and unchecked unions.

5.2 A brief overview of the Pivot

The Pivot is a general framework for the analysis and transformation of C++ programs[13]. The Pivot is designed to handle the complete ISO C++, especially more advanced uses of templates and including some proposed C++0x features. It is compiler independent.

There are lots of (more than 20) tools for static analysis and transformation of C++ programs, e.g., [15, 2, 16, 12]. However, few — if any — handle all of ISO Standard C++ [8, 19], most are specialized to particular forms of analysis or transformation, and few will work well in combination with other tools. The design of the Pivot is focused on advanced uses of templates as used in generic programming, template meta-programming, and experimental use of libraries as the basis of language extension. Since (static) types is central to such libraries, the SELL approach requires a representation that deals with types as first-class citizens and allows analysis and transformation based on their properties. In the C++ community, this is discussed under the heading of *concepts* and is likely to receive some language support in the next ISO C++ standard (C++0x) [21, 18, 20].

The central part of the Pivot is a fully typed abstract syntax tree called IPR (*Internal Program Representation*):

To get IPR from a program, we need a compiler — only a compiler "knows" enough about a C++ program to represent it completely with syntactic and type information in a useful form. In particular, a simple parser doesn't understand types well enough to do a credible general job. We interface to a compiler in some appropriate (to a specific compiler) and minimally invasive fashion. A compiler-specific IPR generator produces IPR on a per-translation-unit basis.

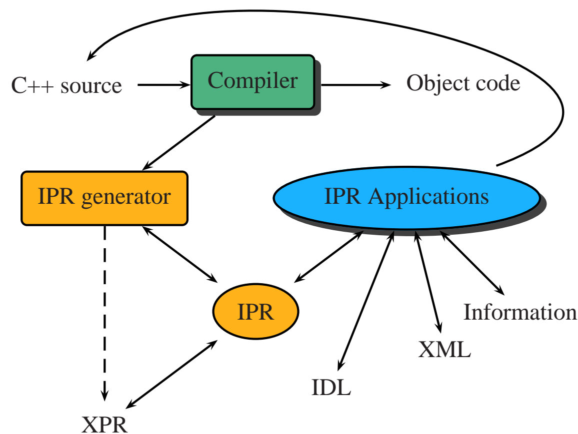


Figure 1. An overview of *The Pivot* infrastructure

Applications interface to "code" through the IPR interface. So as not to run the compiler all the time and to be able to store and merge translation units without compiler intervention, we can produce a persistent form of IPR called XPR (*eXternal Program Representation*).

The IPR is complete and arguably minimal. Traversal of C++ code represented as IPR can be done in several ways, including "ordinary graph traversal code," visitors [6], iterators [19], or tools such as Rose [15]. The needs of the application — rather than the IPR — determines what traversal method is most suitable.

Currently, the Pivot does not support an annotation language. Pivot programs can annotate IPR nodes, but there is no facility for the programmer to embed annotations in the C++ source text. Providing such a facility is easy, but once programmers starts to depend on such annotations, they have created a new special-purpose language. We want to explore how much can be done with the SELL approach, relying only on standard conforming C++ source text.

6 Examples of SELLs

The proof of the pudding is in the eating, but this is not a paper presenting you with a SELL for use; it is a presentation of the general idea of SELLs. Therefore, I present only details that will illustrate the idea of a SELL, not complete SELLs.

6.1 Safe C++

C++ inherits a host of opportunities for type violations from C and adds a few of its own. It is possible — and not very hard — to write type-safe code in C++. However, it is not easy to know that no type violations exist in a program, especially in a large program written and maintained by many programmers with a variety of backgrounds and a variety of ideas of what constitutes safe code. So, how would we support a type-safe dialect of C++ that maintains the essential expressiveness and efficiency of C++? In particular, we want to be sure that there are no type violations in the code. We can only be really sure if we can provide a tool (or combination of tools) that will detect all violations. In the absence of tools, we must rely on humans to follow rules. That would probably be better than the state of the art in most software development organizations, but it would only be second best.

Consider the major insecurities in C++ code:

1. Buffer overruns — i.e., reading or writing outsider the range of an array
2. Dereferencing an uninitialized pointer, a zero-valued pointer, or a pointer to a deleted object
3. Misuse of a union — i.e., write a union variable as one type and read it as another
4. Misuse of a cast — e.g., cast an int to a pointer type where no object of that type exist where the new pointer points
5. Misuse of `void*` — e.g., assign an `int*` to a `void*` and cast that `void*` to a `double*`
6. Deleting an object twice, not deleting an object after use, or using a pointer after deletion.

The obvious approach for avoiding these problems is to provide a library (or a set of libraries) that saves the programmer from having to use these error-prone features. For example, instead of using arrays, the programmer can use a range-checked `vector` and instead of a `union` a user can use a tagged `union` or an `Any` type. Casts (with exception of the dynamically type-safe `dynamic_cast`) and `void*`s are rarely useful outside low-level and easily encapsulated uses, so they can simply be avoided. If we use counted pointers, memory leaks won't happen (depending on how cyclic data structures are handled). Since pointers are checked, we don't access through invalid pointers and double deletions are easily detected.

Basically, errors that cannot be detected until run-time are systematically turned into exceptions, making *Safe C++* a dynamically type safe language. Exceptions may not be your favorite language feature, but they are useful in most contexts and are universally used for reporting run-time type violations in languages deemed type-safe.

So, we can fairly easily write code that doesn't suffer from the obvious type-safety problems. What is outlined here is a SELL where the superset is created by adding checked `vector`s, "smart" checked pointers, a tagged `union` (or an `Any` type). However, nothing has been gained if users persist using the unsafe-features in unsafe ways. For example, we can write safe code, but someone might just do something like this:

```
double* horrible(int i)
{
  int v[80];
  char* p = new char[200];
  double* q = new double[200];
  Shape* pc = new Circle(Point(10,20),20);
  delete[] p;
  p[100] = 'c';
  p[i] = 'x';
  v[100] = 666;
  pc->rotate(45);
  pc->draw();
  f(pc);
  void* vp = v;
  delete vp;
  delete[] p;
  return q;
}
```

Obviously, the subsetting (enforcement) part of the SELL design must be to detect and eliminate the unsafe uses of the host lan-

guage. Please note that the tool that does that must distinguish between the use of the "banned" features or uses of features within the implementation of the extensions and direct use by the user. In this case, a dumb tool (such as a compiler option) banning all uses of pointer would prevent the use of `vector` that uses pointers internally. Instead, we could use the Pivot to catch only the uses of pointers outside our supporting classes. That done, our code would have to be rewritten to look something like:

```
unique_ptr<vector<double>> messy(int i)
{
  vector <int> v(80);
  string p(200);
  vector<double> q(200);
  scoped_ptr<Shape> pc(new Circle(Point(10,20),20));
  p[100] = 'c';      // ok
  p[i] = 'x';        // checked at run time
  v[100] = 666;      // caught at run time
  pc->rotate(45);
  pc->draw();
  f(pc);
  return unique_ptr<vector<double>>(q);
}
```

This is much better (ignoring the messy use of "magic constants"), but *Safe C++* could have problems for real-world programming in many areas where C++ is used: We have not dealt with performance and compatibility. Actually, this code hints of a very significant concern for performance in the library design: `scoped_ptr` deletes its object at the end of scope and prevents `f` from keeping a reference to that object. Similarly, `unique_ptr` cooperates with `vector` to ensure that the elements of `q` are transferred out of `messy` and not destroyed as part of `q` upon exit. We didn't just rely on counted pointers of a garbage collector to deal with resource problems.

Using the Pivot, we could do better, though. By default, both uses of `pc` in `messy` must be checked for validity (assuming that a `scoped_ptr` can be a null pointer). However, a bit of simple flow analysis can eliminate the second check, and a slightly more clever analysis will reveal that no checking is actually necessary: We can see that `pc` has been properly initialized and not assigned to — and so can the Pivot. This kind of analysis has been used experimentally for private languages and dialects [9]. Given the Pivot, we can apply this for a library or for "raw C++."

Compatibility is a harder problem. What if `f` is not known to be safe? What if we can't rewrite or recompile all the code of a system? What if layout compatibility of some data structures is required? *Safe C++* as presented here is just an illustration, not a full-blown SELL.

6.2 Parallel C++

With the emergence of cheap multiprocessors, clusters, and multicore chips, concurrency is increasingly important. Many languages and dialects have been designed to address the concurrency needs of high-performance scientific computing. Here I will build on a library, STAPL [1] [14], that offers parallel operations on containers in the spirit of the STL. For example:

```
void f(pvector<double>& v)
{
  prange<double> r = find_all(v.range(),criteria);
  sort(r);
  cout << r;  // ordinary serial output of elements
}
```

Imagine that `v` has 500 million elements and that the program runs on a serious supercomputer, such as Blue Gene\L[3] (where STAPL is in fact used). The `find_all` will execute in parallel on as many processors as the STAPL run-time system deems reasonable finding elements that meet `criteria`. If `find_all` finds lots of elements, then `sort` will also use many processors.

Here we have a sophisticated library combined with an even more advanced run-time support system. What can the Pivot do to help? For starters, it can produce the information that the run-time support system needs to function well. Secondly, it can provide classical flow analysis and aliasing information. Finally, it can be programmed to recognize usage patterns to allow algorithm substitution (as in the initial matrix algebra example) and alert the programmer to likely problems or opportunities.

6.3 Real-time C++

The problems of real-time code for embedded systems combine concerns for correctness, reliability, and performance in constrained circumstances. Some problems and solutions overlap with those of *Safe C++* but others are unique in that they require that every operation is performed in a known constant time (or less). Naturally, not all real-time and embedded systems are written under this Draconian rule, but let's see how we can address those that are. Some C++ operations become unusable:

1. free store (general `new` and `delete`)
2. exceptions (assuming inability to easily predict the cost of a `throw`)
3. class hierarchy navigation (`dynamic_cast` in the absence of a constant time implementation [7])

First, we add a suitable support library:

1. a fixed size `Array` class (no conversion to pointer, knows its own size)
2. some safe pointer classes
3. memory allocation classes that guarantee constant time allocation (and deallocation if allowed) — pools, stacks, etc.
4. ...

Next, we use the Pivot to eliminate dangerous operations (as listed in 6.1) from user code.

In principle, this will do the job. However, we can do more. For most programs of this sort, we can do whole-program analysis. Such programs tend to be relatively small and not allow dynamic linking. Thus, the Pivot could be used to allow exceptions for error reporting: we can verify that every exception is caught and calculate the upper bound for each throw. This is a special — and especially hard — example of using a tool to verify that resource consumption is within acceptable bounds.

In general, there is lots more that the Pivot can do in the context of embedded systems. Some depends on a specific application, so the boundary between SELL and application support blurs. For example, it is not uncommon for an embedded program to be more permissive about the facilities that can be used during a startup phase. The SELL can define what "startup" means (e.g., called from `start_up`) and only apply the stringent rules outside that.

7 Conclusions

The first half of this paper outlines the problems facing programmers providing and using a special-purpose language defined in the most common ways: as a separate language, as compiler options, as libraries, using a preprocessor for a general-purpose language, and as a dialect. The picture painted is bleak, leading to a suggested alternative: *Semantically Enhanced Library Languages* (SELLs) The SELL approach offers a practical and economical alternative to the more common ways of implementing extensions, dialects, and special-purpose languages. By using libraries, it limits the problems with compatibility and tool chains. By adding tool support, it enhances the appeal of libraries.

8 Acknowledgments

Thanks to Gabriel Dos Reis and my students for reading drafts of this paper and making many detailed suggestions. Also thanks to the anonymous reviewers for making more major suggestions, some of which will require several more papers.

9 References

[1] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, Lawrence Rauchwerger: *STAPL: An Adaptive, Generic Parallel C++ Library* In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), pp. 193-208, Cumberland Falls, Kentucky, Aug 2001.

[2] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, Eelco Visser: *Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs*. `http://www.codeboost.org/`.

[3] IBM: `http://www.research.ibm.com/bluegene/`.

[4] William R. Mark, et al: *Cg: A System for Programming Graphics Hardware in a C-like Language*. Proceedings of SIGGRAPH 2003.

[5] M. Fernandez, et al: *SilkRoute: A framework for publishing relational data in XML*. ACM Trans. Database Syst. 27(4): 438-493 (2002)

[6] Erich Gamma, et al: *Design Patterns*. Addison-Wesley, 1994.

[7] Michael Gibbs and Bjarne Stroustrup: *Fast Dynamic Casting*. Software—Practice & Experience. Vol 35, Issue 686. 2005.

[8] International Organization for Standards, *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd ed., 2003. Wiley 2003. ISBN 0-470-84674-7.

[9] Trevor Jim, et al: *Cyclone: A Safe Dialect of C*. USENIX Annual Technical Conference, pages 275–288, Monterey, CA, June 2002.

[10] Lengauer, et al: *Domain-specific program generation*. Revised papers from Dagstuhl seminar. March 2003. LNCS 3016.

[11] Diane J. Litman, Anil K. Mishra, and Peter F. Patel-Schneider: *Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules*. Proc. 12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97). October 1997. `http://www.research.att.com/sw/tools/r++/` and `http://www.bell-labs.com/project/r++/`.

[12] George C. Necula, et al: *CIL: Intermediate Language and Tools for Analysis and Transformation*. `http://manju.cs.berkeley.edu/cil/`.

[13] The pivot is a program analysis and transformation infrastructure being developed at Texas A&M University.

[14] Steven Saunders, Lawrence Rauchwerger: *ARMI: An Adaptive, Platform Independent Communication Library* In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP), pp. 12, San Diego, CA, Jun 2003. `http://parasol.tamu.edu/groups/rwergergroup/research/stapl/`.

[15] Markus Schordan and and Daniel Quinlan. *A Source-to-Source Architecture for User-Defined Optimizations*. In Proc. of the Joint Modular Languages Conference (JMLC'03), Volume 2789 of Lecture Notes in Computer Science, pp. 214-223, Springer Verlag, June 2003. (Rose).

[16] S. Schupp, D. P. Gregor, D. R. Musser, and S.-M. Liu. *Semantic and behavioral library transformations*. Information and Software Technology, 44(13):797 810, October 2002. (Simplicissimus).

[17] Jeremy G. Siek Andrew Lumsdaine: *The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra*. ISCOPE'98, vol. 1505 of Lecture Notes in Computer Science, 1998.

[18] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. *Concept for C++0x*. Technical Report N1758=05-0018, ISO/IEC SC22/JTC1/WG21, January 2005.

[19] B. Stroustrup, *The C++ Programming Language*, special ed., Addison-Wesley, 2000. ISBN 0-201-70073-5 .

[20] B. Stroustrup: *The design of C++0x*. The C/C++ Users Journal. May 2005.

[21] B. Stroustrup, G. Dos Reis: *A concept design*. Technical Report N1782=05-0042, ISO/IEC SC22/JTC1/WG21, April 2005.

[22] B. Stroustrup: *Examples of C++ applications*: `http://www.research.att.com/~bs/applications.html`. *Some C++ compilers*: `http://www.research.att.com/~bs/compilers.html`.

[23] Todd Veldhuizen: *Arrays in Blitz++* ISCOPE'98, vol. 1505 of Lecture Notes in Computer Science, 1998.

[24] Wilson and Lu (editors): *Parallel programming using C++*. Addison-Wesley. 1996. ISBN 0-262-73118-5.

DMTL: A Generic Data Mining Template Library *

Mohammad Al Hasan, Vineet Chaoji, Saeed Salem,
Nagender Parimi, Mohammed J. Zaki
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
{alhasan,chaojv,salems,parimi,zaki}@cs.rpi.edu

Abstract

FPM (Frequent Pattern Mining) is a data mining paradigm to extract informative patterns from massive datasets. Researchers have developed numerous novel algorithms to extract these patterns. Unfortunately, the focus primarily has been on a small set of popular patterns (itemsets, sequences, trees and graphs) and no framework for integrating the FPM process has been attempted. In this paper we introduce DMTL, a generic pattern mining library which fuses theoretical concepts from formal concept analysis and generic programming. It provides a framework that allows mining a large spectrum of patterns. We express each pattern in terms of its relational properties. Describing patterns based on their properties results in a pattern concept hierarchy. This hierarchical model is implemented using principles from generic programming. In this paper, we describe our design considerations and the subsequent implementation. Some of the challenges faced in terms of language features have also been highlighted. Apart from using the library in its entirety, we believe that some of its components, such as isomorphism checking, can be used independently. These components can definitely enrich the existing functionality provided in some of the popular libraries such as the Boost Graph Library.

1 Introduction

Frequent pattern mining (FPM) is a data mining paradigm to extract informative patterns in massive datasets. Its applications are growing enormously, aided by the availability of high computation power, cheap massive storage, and improved technology for extraction and distribution of data. Researchers have successfully applied FPM to a diverse set of problems in the areas of market basket analysis [1], bioinformatics [27, 26], web mining, fraud detection [4], scientific and medical data mining, etc. In many of these application domains, FPM is not the core component. Hence, availability of the FPM library would allow researchers to save significant effort and would enable them to focus on their core competence. FPM research discovers *patterns* that conceptually represent relations among discrete objects. Depending on the complexity of these relations, different types of patterns originate. The most common type of patterns are sets, where the relation is the co-occurrence of objects. A well known example of the set pattern is a supermarket transaction dataset; the set of items that are bought together by a customer is of interest to the business strategists. Next, there are sequence patterns, where co-occurrence of objects is augmented by the presence of an order between them. Examples include time-series data in financial markets, genome sequence data in bioinformatics, etc. Data mining researchers also work with tree and graph patterns. In tree patterns the object relationship evolves in a hierarchical manner, and in graph patterns the relationship is mostly arbitrary. Mining web log data, XML or semi-structured data are examples of tree mining, and mining chemical compounds for drug discovery is an example of graph mining.

1.1 Related Work

Although FPM is a very mature research area, development of an FPM library has mostly been ignored. Since the commencement of FPM research with the legendary *apriori* itemset mining paper [1] over a decade ago, several hundreds different scholarly articles have been published. Some proposed algorithmic improvements, some covered different variations of FPM problems, such as maximal frequent [2] or closed frequent pattern mining [13] and some developed algorithms for mining new patterns, like DAG (Directed Acyclic Graph), Free Tree [3], etc. Several others demonstrated the potential of FPM algorithms by applying them to new fields, like bioinformatics, operations research, intrusion detection, etc. No real effort has concentrated on developing a library targeting different FPM tasks. The closest works are MLC++ [10] and Weka [20]. The former is a collection of classification algorithms. The latter is a general purpose Java library for different data mining algorithms that includes only itemset mining. Besides these, there are some independent application programs developed by researchers in academia, mostly to evaluate the correctness and performance of their proposed mining algorithms. But they are very specific, run on a selected format of datasets and are in no way suitable as a library component. They do not offer any standard interface for end users. A collection of such algorithms specifically for itemset mining is available from the FIMI [6] web site. Moreover, several practical machine learning software, bioinformatics search tools, etc., employ FPM as the core mining engine, for which they usually write their specific FPM programs. The unavailability of a generic FPM library thus wastes enormous time and computation resources for programmers and researchers.

We developed DMTL (Data Mining Template Library), a frequent pattern mining library, that provides a unified interface to mine a range of patterns. Currently the library has implementations for mining four key patterns—itemset, sequence, tree and graph—but the framework provides the scope to mine new patterns also. DMTL adopts a generic design, inspired by the state-of-the-art generic libraries such as the C++ Standard Template Library (STL) [16, 11] and Boost Graph Library (BGL) [15], and hence it provides widespread usability without compromising on efficiency. The library is generic with respect to the following aspects:

*This work was supported in part by NSF CAREER Award IIS-0092978, DOE Career Award DE-FG02-02ER25538, NSF grant EIA-0103708, and NSF grant EMT-0432098.

- Pattern to be mined.
- Input data source and format.
- Data structure to be used in the mining algorithm.
- Storage management.
- Mining algorithm/approach.

1.2 Contributions

The major contributions of our work towards the data mining community are as follows:

- DMTL offers algorithms for different pattern mining tasks in a unified platform. To the best of our knowledge this is the first effort of this kind in data mining.
- DMTL offers flexible interfaces to each of the algorithms, including each of its sub-tasks so that it is very simple for end users to use it as a library component in their software development.
- DMTL is extensible; new patterns can be mined with very minimal effort from the end user. Users just need to define some template parameters to ensure that the library selects the proper mining algorithm to mine that pattern successfully. Some additional specialized code may be required for efficiency reasons.

We also believe this work contributes to the library development community in the following ways:

- DMTL adopts the generic software development approach using C++ templates. Due to the limitation imposed by the programming language, it is still very difficult for programmers to design generic software. Few books [11, 15] are available that describe an implementation of a generic library. We believe that the design of DMTL could be an example for other generic library developers to follow.
- Apart from its ultimate purpose of discovering frequent patterns, our library provides several stand-alone utilities for various patterns. This primarily includes the isomorphism checking functionality for different patterns. We believe that these features can complement the features provided in BGL.
- While implementing DMTL, we faced numerous challenges, mostly related to programming language support for generic software development. Most of these issues have already been identified by several researchers [14, 17], but our work stands as another practical example of those limitations.
- DMTL uses several template tricks, which we think could be tremendously useful for any generic software developer.

2 Pattern Mining Preliminaries

The problem of mining frequent patterns can be stated as follows: let $\mathcal{N} = \{x_1, x_2, \ldots, x_{n_v}\}$ be a set of n_v distinct nodes or vertices. A pair of nodes (x_i, x_j) is called an edge. Let $\mathcal{L} = \{l_1, l_2, \ldots, l_{n_l}\}$, be a set of n_l distinct labels. Let $L_n : \mathcal{N} \to \mathcal{L}$, be a node labeling function that maps a node to its label $L_n(x_i) = l_i$, and let $L_e : \mathcal{N} \times \mathcal{N} \to \mathcal{L}$ be an edge labeling function, that maps an edge to its label $L_e(x_i, x_j) = l_k$.

A *pattern* P can be represented as the pair (P_V, P_E), with labeled vertex set $P_V \subseteq \mathcal{N}$ and labeled edge set $P_E = \{(x_i, x_j) \mid x_i, x_j \in P_V\}$. The number of nodes in a pattern P is called its *size*. A pattern of size k is called a k-pattern, and the class of frequent (as defined below) k-patterns is referred to as F_k. Given two patterns P and Q, we say that P is a *sub-pattern* of Q (or Q is a *super-pattern* of P), denoted $P \preceq Q$, if and only if there exists a label-preserving isomorphism from P to Q; that is, iff there exists a 1-1 mapping f from nodes in P to nodes in Q, such that for all $x_i, x_j \in P_V$: i) $L_n(x_i) = L_n(f(x_i))$, ii) $L_e(x_i, x_j) = L_e(f(x_i), f(x_j))$, and iii) $(x_i, x_j) \in P_V$ iff $(f(x_i), f(x_j)) \in Q_V$. In some cases we are interested in *embedded* sub-patterns. In embedded patterns we modify condition iii) above to allow an edge (x_i, x_j) in P provided $f(x_i)$ and $f(x_j)$ are connected in Q. In other words, P is an embedded sub-pattern of Q if P is a sub-pattern of the transitive closure of Q. If $P \preceq Q$ we say that P is contained in Q or Q contains P.

A database $\mathcal{D}$ is just a collection of patterns (objects, in database terminology). Let $\mathcal{O} = \{o_1, o_2, \ldots, o_{n_o}\}$ be a set of n_o distinct *object identifiers*. An object has a unique identifier, given by the function $O(d_i) = o_j$, where $d_i \in \mathcal{D}$ and $o_j \in \mathcal{O}$. The number of objects in $\mathcal{D}$ is denoted by $|\mathcal{D}|$. The *absolute support* of a pattern P in a database $\mathcal{D}$ is defined as the number of objects in $\mathcal{D}$ that contain P, given as $\pi^a(P, \mathcal{D}) = |\{P \preceq d \mid d \in \mathcal{D}\}|$. The *(relative) support* of P is given as $\pi(P, \mathcal{D}) = \frac{\pi^a(P, \mathcal{D})}{|D|}$. A pattern is *frequent* if its support is greater than a user-specified minimum support (*min_sup*) threshold, i.e., if $\pi(P, \mathcal{D}) \geq min_sup$. A frequent pattern is *maximal* if it is not a sub-pattern of any other frequent pattern. A frequent pattern is *closed* if it has no super-pattern with the same support. The frequent pattern mining problem is to enumerate all the patterns that satisfy the user-specified *min_sup* frequency requirement (and any other user-specified conditions).

The main observation in FPM is that the sub-pattern relation $\preceq$ defines a partial order on the set of patterns. If $P \preceq Q$, we say that P is more general than Q, or Q is more specific than P. The second observation used is that if Q is a frequent pattern, then generally all sub-patterns $P \preceq Q$ are also frequent.[1] More important is the converse, i.e., if P is infrequent and $P \preceq Q$ then Q shall also be infrequent (follows from the anti-monotonicity of frequency). The *prefix* of a pattern of size k is a sub-pattern that consists of the first $k-1$ nodes of the pattern. For efficiency reasons, many FPM algorithms group (at least conceptually) patterns having the same prefix into a *prefix-based equivalence class*. The various FPM algorithms differ in the manner in which they search the pattern space.

3 Generic Aspects of DMTL

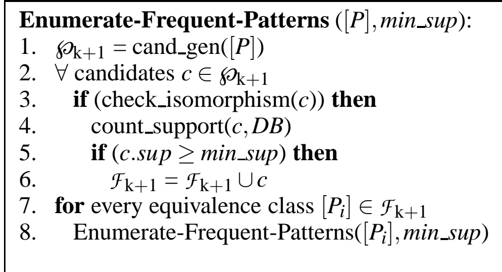
In this section we outline the generic aspects of the Data Mining Template Library.

3.1 Generic Mining Algorithm

While implementing mining algorithms for different patterns, we noticed that they exhibit considerable similarity, which suggests developing a common framework for implementing them. Figure 1 outlines a generic pattern mining algorithm (pseudo-code) that applies to all commonly explored patterns. In the algorithm (not shown in the figure), k is initialized to zero and DB represents a global database. Similarly, other related pattern mining algorithms (closed or maximal pattern mining) also conform closely with this outline. The algorithm is broken down into the major sub-tasks which includes **candidate generation**, **isomorphism checking** and

[1] Note that this property does not hold for induced patterns.

support counting (explained in detail in the implementation section). By implementing generic functions for these sub-tasks, we retain the abstraction shown in this pseudocode. The overall idea of the algorithm is as follows: the mining process searches incrementally in the pattern space by iteratively applying these sub-tasks in each iteration to enumerate patterns of size 1, 2, and so on. Each iteration discovers frequent patterns sized one greater than the previous till no further frequent patterns exist in the database. The example in figure 2 demonstrates how the generic algorithm works for itemset mining. The database on top left corner of the figure has 4 transactions. Each row contains a collection of items separated by commas. We want to perform itemset mining on this dataset with an absolute minimum support value of 3. The same database is also shown in its vertical format (explained later in subsection 3.1.2). This representation is important in the vertical mining approach. The algorithm first finds all the size-1 frequent itemsets, by making a single database scan. The frequent items from the dataset with a support value 3 or more are A, C, T and W, which are shown in the oval to the right of the dataset. Each of these items is present in at least 3 transactions. Now, the candidate generation step generates six size-2 candidates by joining items from this set. The possible candidates here are shown in the rectangle under the oval. Note that the joining process in itemsets automatically eliminates duplicates. For joining complex patterns (joining two graphs), this may not be the case, and we need to employ isomorphism checking to ensure that each candidate pattern is generated exactly once. Finally, the support counting step counts the support of each of the candidates from the database. This step drops the itemset AT, as it appears in only $2(<3)$ transactions. The algorithm iterates until the size-k patterns are found. All frequent itemsets produced by this algorithm are shown in the figure. For other patterns, the algorithm follows the exact same approach as detailed in this example.



Enumerate-Frequent-Patterns $([P], min_sup)$:
1. $\wp_{k+1}$ = cand_gen($[P]$)
2. $\forall$ candidates $c \in \wp_{k+1}$
3. **if** (check_isomorphism(c)) **then**
4. count_support(c, DB)
5. **if** ($c.sup \geq min_sup$) **then**
6. $\mathcal{F}_{k+1} = \mathcal{F}_{k+1} \cup c$
7. **for** every equivalence class $[P_i] \in \mathcal{F}_{k+1}$
8. Enumerate-Frequent-Patterns($[P_i], min_sup$)

Figure 1: Pattern Mining Algorithm

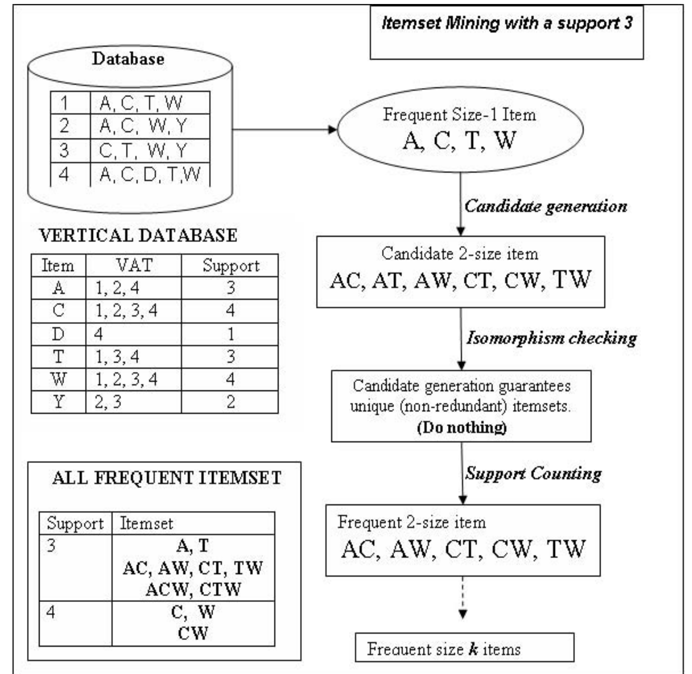


Figure 2: Itemset Mining Example

The sub-tasks of a generic mining algorithm that we referred to in the above two sections can be developed by using generic algorithms expressed with C++ function templates. For example, the **candidate generation** step takes two parent patterns of type T and generates one or more candidate patterns of type T. Here, the algorithm strictly requires that both the input arguments, together with the output argument, are of the same type T (e.g., we cannot join a set pattern with a tree pattern to produce a tree candidate pattern). The **isomorphism checking** algorithm takes two input arguments of same type T (a pattern type) and produces a boolean value to indicate whether the arguments are isomorphic patterns or not. The **support count** algorithm takes one input argument of pattern type T, counts its frequency in the entire database and returns an integer value.

In all the above three generic algorithms, the type T models a pattern concept. It has the following requirements:

1. T defines an object that relates some elements.
2. T must adhere to a structure that is defined by a collection of relational properties.
3. T defines a $\leq$ operator.
4. Associated with type T there exists a pattern-iterator, which is used to iterate through the elements of the pattern.

All commonly known patterns in data mining, like set, sequence, tree or graph are refinements of a pattern concept. The relational properties of a pattern concept that we refer to as *pattern properties* in DMTL are explained in the following subsection.

3.1.1 Pattern Properties

In section 2, we defined patterns in terms of graph abstraction. The choice of graph, indeed comes naturally, since all the patterns are, in a way, specializations of a graph pattern (a set is a special case, which we considered as a graph without any edge). Hence, a graph can represent all the patterns both conceptually and implementation-wise. Using graph implementation for more simpler patterns, like set, sequence or tree introduces inefficiency in the mining algorithm, however, the concept of *pattern property* provides a novel solution to this dilemma. In the implementation section, we explain the way we use *pattern properties* to ensure a generic algorithm that does not compromise efficiency. Here, we explain the different pattern properties that we used.

Relational properties that a pattern type T must conform to, are indeed the graph properties. These properties imposes constraints on graph to formulate patterns like, tree, sequence etc. We analyzed the pattern space and found that the following properties are sufficient to describe the most common patterns, but nevertheless, additional properties may be added seamlessly. The properties are themselves categorized depending on the elements (nodes, edges,

etc.) of a graph on which the constraints are imposed.

1. **Edge Relation** The edge set E_g is defined as $E_g \subseteq V_g \times V_g$. Under edge relation category we considered the following properties.
   - **no-edge** Elements in the patterns are not connected with any edge.
   - **directed** Elements in the patterns are connected with directed edge. To put it in another way, we can say, they are asymmetrically related.
   - **undirected** Elements in the patterns are connected with symmetric edges.
   - **cyclic** A pattern is cyclic if at least one vertex is reflexive on edge relation in the transitive closure of the pattern, otherwise the pattern possess the acyclic property.
2. **Vertex**
   - **order** The *ordered* property imposes an ordering on the neighbors of a vertex, or else the pattern is said to be unordered. Ordering is usually relevant for the tree pattern only.
3. **Degree**
   - **indegree_lte_one** This property constrains all vertices of a graph to have indegree ≤ 1.
   - **outdegree_lte_one** This property constrains all vertices of a graph to have outdegree ≤ 1.
4. **Label**
   - **unique_label** This property requires the labeling function to be one-to-one (injective). Each vertex thus maps to a unique label (a common example of such a pattern is an itemset).

3.1.2 Mining Properties

So far, we discussed that the generic mining algorithm that DMTL advocates can mine any pattern belonging to a pattern concept. But, in data mining research several variations of the core generic mining algorithms exist, by varying the manner in which we perform its sub-tasks. We represent those variations in terms of *mining property*; a user can choose a collection of such mining properties to select the exact kind of algorithm that (s)he would like to choose for the mining process. It is worth noting that, the mining properties are independent from the pattern properties. An analysis of existing FPM tasks revealed the following mining properties that we mention below. As with pattern properties, new mining properties can also be added effortlessly.

1. **Join-type** This category influences the candidate generation phase, in which potentially frequent pattern are generated. During candidate generation, the algorithm typically constructs a new pattern by *joining* two parent patterns. The nature of this join is a property itself. A suitably correct algorithm has to be provided for the chosen property.
   - $\mathbf{F_k} \times \mathbf{F_1}$ A $(k+1)$-length pattern is constructed by joining a k-length pattern with a unit length pattern.
   - $\mathbf{F_k} \times \mathbf{F_k}$ A $(k+1)$-length pattern is constructed by joining two k-length patterns. This join is usually more efficient since it generates fewer infrequent candidates.
2. **Support-counting** This category specifies how the support of a candidate pattern is determined. Two common approaches are:
   - **horizontal** Indicates that the support for a candidate pattern shall be determined by counting its occurrences in the database, testing against each database object. This method usually involves significant I/O overhead for large databases.
   - **vertical** In this approach, support for a pattern is determined from what is called a *vertical representation* of a pattern [22]. This vertical representation for a pattern is a list of transactions in which the pattern occurs and is commonly referred to as *Vertical Attribute Table* (VAT). A vertical database lists all the patterns along with their VATs. Figure 2 shows a vertical database in the table titled "Vertical Database". Support counting using a vertical database is typically faster as it reduces I/O cost.
3. **Transitivity** This category indicates if embedded occurrences of a pattern should be considered in its support counting.
   - **induced** Only induced pattern occurrences are counted.[2]
   - **embedded** Transitive closures on the edge relation E are included in the support as well. The transitivity leads to discovery of embedded occurrences of the pattern.

3.2 Generic Storage Manager

Database (back-end) support is an integral part of any pattern mining task. Since pattern mining datasets are typically large in size, back-end management becomes crucial to achieving an efficient implementation. Sometimes a dataset does not even fit in main memory, so part of it needs to be saved on the disk for the algorithm to continue. Since back-end access is tightly embedded in the mining algorithm, it is very difficult for the user to modify the back-end to obtain scalability or persistence.

DMTL's implementation of back-end database support is generic, through a generic storage manager class. Following the STL iterator concept, we decoupled the back-end database from the algorithm using iterators. Any access to the database is done only through the iterators. We also implemented three different storage managers; all provide iterator classes. Discussion about each of them is given in the implementation section.

3.3 Generic Input Data Source

DMTL is implemented with an objective to be widely applicable. However, the format of the input dataset is different for different application domains. For instance, in supermarket transaction databases, items are usually represented by numeric identifiers, whereas in bioinformatics, items may use string representations for protein or DNA sequences. DMTL takes care of these kinds of dataset irregularities by implementing a generic tokenizer, which is templatized with various arguments to adapt to a wide variety of input datasets.

[2] Note that for graphs we actually mine connected sub-graphs, and not only induced sub-graphs.

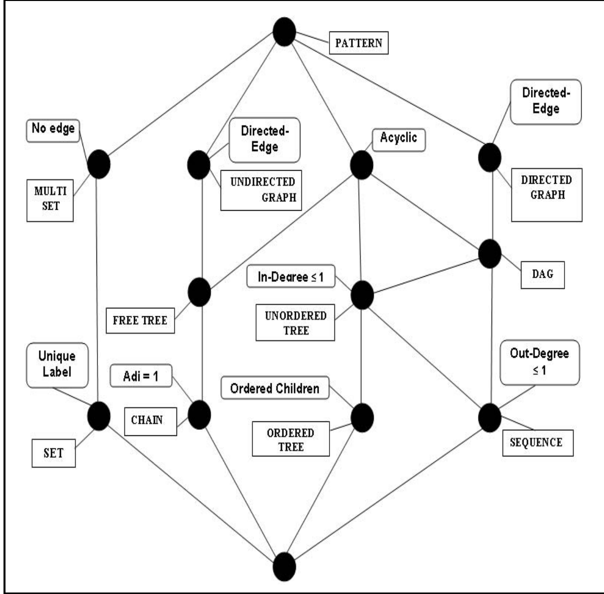


Figure 3: Pattern Property Concept Lattice

4 Pattern Property Concept

The generic design of DMTL mining algorithms for all patterns based on the *pattern property* has a foundation in Formal Concept Analysis (FCA) [5]. We explain this next.

4.1 Formal Concept

DEFINITION 1. *A **formal context** $(K) := (G, M, I)$ consists of two sets, G and M, and a relation I. The elements of G are called the **objects** and the elements of M are called the **attributes** of the context. In order to express that an object g is in the relation I with an attribute m, we write gIm or $(g,m) \in I$ and read it as "object g **has** attribute m."*

DEFINITION 2. *For a set $A \subseteq G$ of objects we define*

$$A' := \{m \in M \mid gIm,\ \forall g \in A\}$$

(the set of attribute common to the objects in A). Correspondingly, for a set B of attributes we define

$$B' := \{g \in G \mid gIm,\ \forall m \in B\}$$

(the set of objects which have all the attributes in B.)

DEFINITION 3. *A **formal concept** of the context (G, M, I) is a pair (A, B) with $A \subseteq G, B \subseteq M, A' \subseteq B$ and $B' \subseteq A$. We call A the* **extent** *and B the* **intent** *of the concepts (A, B). $\mathfrak{B}(G, M, I)$ denotes the set of all concepts of the context (G, M, I).*

In DMTL, we consider G as the set of all patterns that we want to mine, M as the set of all pattern properties and I as the relation that a pattern conforms to a property, then (G, M, I) is a context. Now, if $A \subseteq G$ is maximal a collection of patterns, and $B \subseteq M$ is the set of properties that are common to all the patterns in A, then (A, B) is a formal concept of the context (G, M, I).
Example: If A = {*DAG, Sequence, Ordered Tree, Unordered Tree*} is the set of patterns and $B = \{Directed, Acyclic\}$ is the set of properties common to members of A, then (A, B) forms a formal concept. The set A, i.e. the set of patterns, is the extent of the concept and B, the set of properties, in the intent of the concept.

The concept in generic programming adheres with definition 3, if the objects equate with abstractions (types, in particular) and the attributes with requirements. In [18], Willcock et al. provide a precise definition for concepts, as they are used in practical generic programming. That definition is an extended form of the above definition, where the extensions clarify several issues related to generic software design and programming languages.

4.2 Formal Concept Lattice

DEFINITION 4. *If (A_1, B_1) and (A_2, B_2) are concepts of a context, (A_1, B_1) is called a **sub-concept** of (A_2, B_2), provided that $A_1 \subseteq A_2$ (which is equivalent to $B_2 \subseteq B_1$). In this case, (A_2, B_2) is a **superconcept** of (A_1, B_1), and we write $(A_1, B_1) \leq (A_2, B_2)$. The relation $\leq$ is called the **hierarchical order** of the concepts. The set of all concepts of (G, M, I) ordered in this way is denoted by $\mathfrak{B}(G, M, I)$ and is called the **formal concept lattice** of the context (G, M, I).*

Example: The set of all *pattern-property formal concepts* form a concept lattice as illustrated in Figure 3. In this figure, every node is a formal concept. The corresponding set of objects and attributes of that concept are shown next to it, in boxes with rectangular and rounded edges, respectively. Every box only list those objects or attributes that are not implicitly inherited through the refinement relation (discussed in next paragraph). We can retrieve the entire set of extents (objects) by tracing all paths which lead down from that node. On the other hand, the intents (attributes) can be obtained by tracing all paths leading upward from that node.

If we consider the node labeled with the formal object DAG, it represents a formal concept with objects

$$\{\text{DAG}, \text{Sequence}, \text{Unordered Tree}, \text{Ordered Tree}\}$$

and with properties $\{\text{Acyclic}, \text{DirectedEdge}\}$

4.3 Concept Refinement

DEFINITION 5. *Concept refinement is the process of obtaining a sub-concept from a concept. Adding one or more attributes in the intent removes objects from the extent that do not conform to that property.*

Example: We can refine the concept in the above example by adding one property named `indegree_lte_1`. In the refined concept, the pattern DAG is omitted, as DAG does not conform to this property.

4.4 Concept Refinement in DMTL Design

In our generic library implementation, we employed understanding of formal concept hierarchy to develop mining algorithms that can handle different types of patterns. Any algorithm that works for patterns in a pattern-property concept automatically works for the sub-concept. For patterns in sub-concepts, a list of pattern properties that is passed as template arguments matches partially and automatically invokes the algorithm for the patterns belonging to the immediate super-concept. However, there could exist a more efficient implementation for the patterns in the sub-concept as they might be comparably easier to mine. For those cases, we provide

a more efficient implementation of the algorithm as an overloading of the template function.[3] We discuss the implementation details in the following section.

5 Implementation Issues

This section describes the implementation details of DMTL. Three major subsections cover the architecture, data and algorithms of DMTL respectively.

5.1 Architecture

Figure 4 provides a quick look at the various architectural components (in rectangular boxes) of DMTL. We partitioned the components into two main segments—the *front end* and the *back end*. The front end deals with the core mining process while the back end provides the necessary storage support.

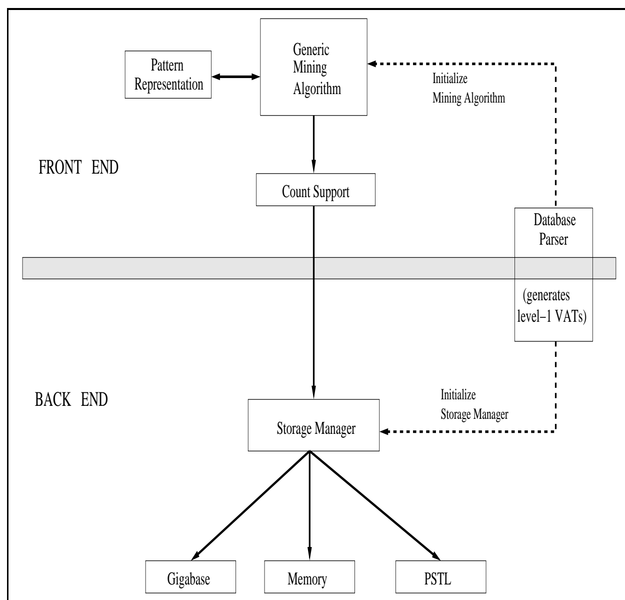


Figure 4: High-level Architecture Diagram of the Data Mining Template Library

5.1.1 Front-end: The Mining Engine

The mining task is initiated with all frequent patterns of length one. This step is performed by reading the data from a source. The source could either be a database, a flat file or another process that is generating the data. This functionality is performed by the *Database Parser* module (see figure 4). Then the generic algorithm generates unique candidate patterns through candidate generation and isomorphism checking, as we explained in section 3.1. The task of finding the support of each candidate pattern is delegated to the back end through the *Count Support* module.

[3]If we were expressing algorithms with classes we would provide the more efficient algorithms as partial template specializations, but in the case of function templates one must currently use overloading instead. Proposals to add partial specialization of function templates to the language standard have been made but to date have not been accepted.

5.1.2 Back end: The Storage Manager

Frequent pattern mining is often performed on very large datasets. Each iteration of the algorithm generates increasingly larger patterns, and the number of candidate patterns also grows enormously (especially, with low support) and does not fit in memory on most machines. In a vertical mining paradigm, associated with each pattern, a VAT also needs to be stored. Most mining algorithms do not provide explicit means of memory management nor is the issue addressed within the algorithm. The DMTL back end is dedicated to storage management, which stores the patterns, VATs, and the associated one-to-one mapping from patterns to their VATs. The back end also determines the support count of candidate patterns and returns it to the front end.

The current state of DMTL has multiple implementations of the back end—memory, Gigabase [9] and PSTL [7]—each one exporting the same interface. The Count Support module can select any one of these by using template arguments. Gigabase is an embedded object relational database which has its own storage management. It also stores elements (patterns, VATs) in its database file. PSTL is a library of persistent containers, akin to STL in its design. PSTL also achieves persistence by maintaining memory-mapped data files. In both the above cases, the mining results and intermediate data (like VATs) are stored on disk and are available for processing at a later point. Thus, DMTL provides an elegant solution when a memory-based back end fails due to enormous growth of data. A flexible interface makes addition of a new storage manager type quite easy. We also considered using third party object stores as storage managers. Lack of flexible libraries for object storage prompted us to develop our own storage manager.

5.2 Data Types

The most vital data in DMTL are the patterns and their associated VATs. Patterns are implemented with a graph structure. Elements of a pattern are the vertex or edge labels of that graph. VATs are implemented using `std::vector`, as they store a list of transaction identifiers. And for the mapping between pattern and VAT, we use `std::map`. However, pattern structure plays the most important role in our generic mining algorithm, so we describe it further in the following section.

5.2.1 Pattern Structure

In DMTL, vertices and edges are the basic structural building blocks of every pattern. The most basic interface for a pattern should thus provide methods for adding labeled vertices and directed edges between vertices. Figure 5 shows the C++ class interface of the pattern concept that we mentioned in 3.1. It consists of the most basic operations expected from a type modeling such concept. A specific pattern (set, sequence, tree, etc.) is defined by enlisting the respective pattern properties (`pattern_props`). The `canonical_code` template parameter maintains a unique code corresponding to each pattern and is employed for isomorphism checking. It also provides binary inequality testing operations that can be used to implement the $\leq$ operator for the pattern concept. The `graph_model` is the underlying data structure used for storing the above representation. A typical example of such a data structure is an adjacency list. This design decision to parameterize the storage type aims at decoupling the pattern storage from the pattern concept, such that an adjacency list based storage could be substituted by a sparse adjacency matrix structure. Our design underlines the fact that loose coupling between key design components is crucial

for the extensibility of a large software system. From the above interface, a sequence such as $A \rightarrow B$ can be constructed by invoking the *add_vertex("A")* method followed by the *add_vertex("B")* and *add_out_edge(v1, v2, e)* methods. The Boost Graph Library

```
template<class pattern_props, class graph_model,
         class canonical_code>
class pattern {

  public:
    typedef vector<V_TYPE> VERTICES;
    typedef typename VERTICES::const_iterator
                                    CONST_VIT;

    bool add_vertex(const V_TYPE& v);
    bool add_out_edge(const V_TYPE& v1,
                      const V_TYPE& v2,
                      const E_TYPE& e);
    bool add_in_edge(const V_TYPE& v1,
                     const V_TYPE& v2,
                     const E_TYPE& e);
    CONST_VIT get_neighbors(const V_TYPE& v);
    CONST_VIT get_rmost_path();
};
```

Figure 5: Pattern Class Interface

(BGL) [15] provides a more complete set of graph representations and graph algorithms. At this moment we have refrained from using BGL's graph representations, primarily to keep the design flexible and open to various possibilities. In the future, we aim to utilize BGL's graph primitives to standardize our library. As seen in

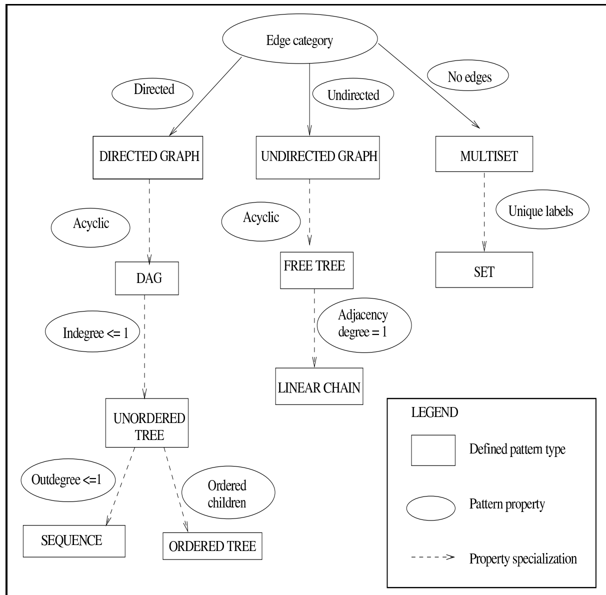


Figure 6: Pattern Hierarchy

figure 3, the specific patterns are instantiations of the abstract pattern concept. Each such concrete concept is represented by a set of properties (or constraints) that define the pattern. For instance, a directed acyclic graph (as the name suggests) has {*acyclic, directed*} as its property set. The notion of having a set of properties to represent a concept is crucial for the implementation of our library. Even though conceptually the properties are considered to be a set, from the implementation perspective we treat them as an ordered list of properties. This ordering of properties is necessary for the compiler to match a specialized pattern to an appropriate super-pattern, if any algorithmic implementation is not available for that specialized pattern. This leads to the pattern hierarchy tree in figure 6. Note that in figure 3, a node can have multiple parents while in the pattern tree each pattern has a single parent. The importance of the single-parent characteristic becomes evident when we realize that selecting a super-pattern would lead to ambiguities in case of multiple super-patterns. Using this pattern hierarchy tree, the ordering of the properties for a pattern is automatically enforced. They are ordered along the path from the root to a pattern node. In a nutshell, figure 3 represents the conceptual (theoretical) side of the pattern mining problem whereas figure 6 represents the practical (implementation) side of the problem.

We had the following goals while constructing the hierarchy of patterns:

1. Abstract out the common aspects between the pattern types and the algorithms,
2. Allow new patterns to be added to the hierarchy by introducing new properties, and
3. Propagate absence of a lower-level concept implementation to a higher-level concept implementation.

The last objective above is a logical extension of using partial specialization (via function template overloading). The presence of a single parent in the hierarchy tree enables finding the right pattern to which control should be dispatched. Our library provides implementations for what we call the four core patterns—sets, sequences, trees and directed graphs. Apart from being the most popular patterns, the core patterns can be considered to mark the complexity classes in frequent pattern mining. Sets are at the simpler end of the spectrum with sequences and trees (in that order) before graphs at the other extreme. The following paragraphs describe the challenges faced in designing the library to achieve the first two goals.

5.2.2 Pattern Properties Implementation

In order to enable dispatching to the appropriate pattern we use the set of pattern properties as template parameters. This set of pattern properties is encapsulated in a **proplist**. Since we model properties as types, the *proplist* is a static list of types provided for collecting properties. It should be noted that such a type list is a static accumulator, i.e., it relies on the template compile-time mechanism and hence incurs no run-time overhead. A type list gives us the flexibility to append properties to it, making the design generic and extensible. The type list was designed by borrowing ideas from two of the C++ Boost libraries—the Boost Graph Library and the Metaprogramming Library [15]. Since it is simply a container of types, the class itself is not complicated and is given in Figure 7. The class `null_prop` is used as the terminator of a type list. In addition to its utility as a type list, the `proplist` possesses the nice feature of facilitating upward propagation of properties. This behavior is demonstrated in Figure 8. To keep the example simple, we have stripped function parameters and return types that are not relevant for the example. In this example, we create property classes and give the prototype of a function that generates candidates from a given pattern. As pointed out above, candidate generation is one of the three tasks a mining algorithm must undertake. In the figure, two prototypes of the `candidates` function are provided—one

```
template<class prop,
         class next_property=null_prop>
class proplist {
  public:
  typedef prop FIRST;
  typedef next_property SECOND;
};
```

Figure 7: proplist Class Interface

```
/// Property class definitions ///
class directed {};
class acyclic {};
class planar {};
class null_prop {};

/// generic function ///
void candidates(const proplist<directed>&);

/// specialized function for DAGs ///
void candidates(const proplist<directed,
                proplist<acyclic> >&);

///// an illustration of how it works /////
proplist<directed> digraph;
proplist<directed, proplist<planar> > planar_graph;
proplist<directed, proplist<acyclic> > dag;

/// Following function call compiles  ///
/// to generic function.            ///
candidates(digraph);

/// Following function call compiles  ///
/// to specialized function.        ///
candidates(dag);

/// Following function call compiles  ///
/// to generic function               ///
candidates(planar_graph);
```

Figure 8: Application of Property Hierarchy

for directed graphs and one for DAGs. DAGs do not possess cycles, hence the specialized `candidates` function does not generate cyclic graphs as candidate DAGs. On the other hand, the generic function generates all possible digraphs, including cyclic ones. This relation between DAGs and directed graphs is reinforced by the pattern hierarchy in figure 6). Hence, as expected, method calls with directed graph and DAG as their input parameter types would invoke the appropriate methods. The `planar_graph` property list is now introduced. It should be noted at this point that the pattern property `planar` is not defined in our library. Hence, it is a new pattern property for representing planar graphs. Let $\mathcal{P}_1$ denote the pattern type, digraphs, and $\mathcal{P}_2$ denote directed, planar graphs. Since the properties defining $\mathcal{P}_1$ are a subset of the properties defining $\mathcal{P}_2$ we can say $\mathcal{P}_1 \preceq \mathcal{P}_2$. As a result a `candidates` method call with `planar_graph` as input parameter will invoke the method with `digraph` as the formal parameter. Had there been a more efficient implementation for planar digraphs, that would have been invoked. To summarize, we have shown how the `proplist` can be used to select the most appropriate implementation and how a new pattern can be easily introduced into the framework.

5.3 Generic Algorithms

The core FPM algorithm shown in Figure 1 was introduced in section 3.1. Even though we do not enforce a pattern to conform to this precise formulation of the mining process, most FPM algorithms (including the ones in our library) conform closely to this outline.[4] The pseudocode in figure 1 is implemented in the `freq_pat_mine` method.

```
template<class PATTERN, class MINE_PROPS,
         class SM_TYPE>
void
freq_pat_mine(const pat_fam<PATTERN>& Fk,
         const pat_fam<PATTERN>&, int& min_sup,
         pat_fam<PATTERN>& freq_pats,
         count_support<MINE_PROPS,
                        SM_TYPE >& cs)
```

The first parameter to this method, `pat_fam`, is a collection of patterns that belong to the same prefix-based equivalence class and can be implemented as an STL vector or a list. The third parameter, `freq_pats`, which is passed by reference, is used to collect the final set of frequent patterns. Our customized containers either retain the same interface as the popular STL containers or are simply wrappers around STL containers. Note that in the above example `PATTERN` is the pattern representation. Hence it is not just a container parameter but is used to pick the most efficient implementation along the pattern hierarchy. The actual template argument could represent any pattern. As the name suggests, the `count_support` class is used for finding the support of the candidate patterns in the dataset. `count_support` is templated on the mining properties and back-end database type. The former is necessary because counting support differs for embedded and induced mining (which is a mining property). The later (`SM_TYPE`) is necessary for querying the appropriate storage manager to find the number of occurrences of a pattern. Let us take a closer look at some of the key steps inside `freq_pat_mine`.

5.3.1 *Candidate Generation*

Pattern types differ in how they generate candidates. However, there does exist significant commonality among the varying pattern types. This was explored by us in a previous work [25]. The `freq_pat_mine` method calls the `join` method to generate new candidates by joining two frequent patterns. The interface for the join method is as shown below:

```
template<class PAT_PROPS,
         class MINE_PROPS,
         class SM_TYPE>
pattern<PAT_PROPS,
        MINE_PROPS,
        SM_TYPE>**
join(const
      pattern<PAT_PROPS, MINE_PROPS,
              SM_TYPE>* pat_i,
     const
      pattern<PAT_PROPS, MINE_PROPS,
              SM_TYPE>* pat_j)
```

[4] FP-tree is another approach for FPM. Since it is not as widespread as the apriori based approach, DMTL does not currently support it.

This method takes two pattern pointers and outputs an array of pattern pointers (an array is chosen, as sometimes more than one pattern is created from the join operation). Note that both the pattern properties and the mining properties are associated with the pattern type. Using pattern properties, the join method chooses the most appropriate algorithmic implementation to perform the join for this pattern type. Note that a join between patterns is associated with an intersection of the corresponding VATs. For example, if a pattern A is a set $\{a,b,c\}$ and another pattern B is a set $\{a,b,d\}$ and their VAT (list of transactions they occur in) are $\{1,4,10\}$ and $\{1,10,12\}$ respectively. A join (set union operation) produces one pattern $\{a,b,c,d\}$, and the corresponding intersection of VATs (set intersection operation) produces $\{1,10\}$, which is the VAT of the new pattern. However, the join method shown here materializes the pattern join only; the associated VAT intersection is done in the back end.

5.3.2 Isomorphism Checking

For itemsets and sequences we can circumvent generating isomorphic patterns by intelligent candidate generation [1, 23]. Essentially, we exploit the lexicographic ordering on the labels to avoid generating redundant patterns. Isomorphism checking can also be avoided for ordered trees by an appropriate candidate generation scheme [24]. However, unordered trees [12], free trees [3] and graphs [21, 8] require isomorphism testing. The isomorphism checker is provided by the `check_isomorphism` method and it is templatized on the pattern properties. Our library provides specialized isomorphism routines for various patterns—directed graphs and unordered trees, to name a few. The isomorphism checker can be used as a stand-alone component and we believe that it could further enrich the isomorphism checking support provided in BGL.

```
template<class PAT_PROPS,
         class MINE_PROPS,
         class SM_TYPE>
bool
check_isomorphism(pattern<PAT_PROPS,
                          MINE_PROPS,
                          SM_TYPE>* cand_pat)
```

5.3.3 Support Counting

The last step in an iteration is to determine the support of candidates, and discard ones that do not pass the *min_sup* (minimum support) criterion. The support counting functionality is supported by the *Count Support* block in figure 4. Since support counting needs to query the back end, this block acts as a liaison between the front end and the back end. The support counting module is common across all the pattern types, since it does not need to know anything about a specific pattern. At the same time the `count` method is independent of the back end since the `count_support` class is templatized on the storage type. The interface for the `count` method is given below:

```
template<class PATTERN>
void
count(PATTERN* p1, PATTERN* p2, int min_sup)
```

As we mentioned under Candidate Generation above, a join of patterns in the front end triggers an associated VAT intersection in the back-end. We provided different back-end implementations, all storing the same VAT but may be in different formats. For example, the VAT stored in the Gigabase database is necessarily different than that stored in the memory back end. Nevertheless, the VAT intersection algorithm is the same. Inspired by STL's design, we used iterator concepts to decouple the algorithm from the actual data structure. Figure 9 shows how iterators hide the data representation from the algorithms. The figure shows the signature of

```
template<typename InIter,
         typename OutIter>
void intersection(pair<InIter, InIter> itr_i,
                  pair<InIter, InIter> itr_j,
                  OutIter cand_vats);
```

Figure 9: Using Iterators with Generic Algorithms

the `intersection` method, which joins two VATs to generate the VATs for new candidate patterns. The first parameter is a pair of iterators pointing to the beginning and end of the container that corresponds to the first VAT. Similarly, the second parameter is for the second VAT. The two iterators use the same `InIter` parameter since patterns have to be of the same type to be intersected. The third parameter represents an output iterator and is used to collect the set of generated VATs. Note that, depending on the pattern, more than one VAT could be generated.

To reiterate, the design of DMTL consists primarily of three challenging components:

1. pattern structure,
2. pattern algorithms, and
3. back end storage facility.

Along with the above key components, the library contains multiple smaller utilities for reading in data from multiple sources, parsing data in multiple formats, and many others.

5.4 Incorporating new patterns

Representing patterns as property-based concepts allows users to introduce new properties, and hence new patterns, with minimal changes to the code. This effectively allows us to mine any type of pattern. This idea of mining arbitrary patterns is novel and extremely desirable in the data mining community. Let us walk through an example to see how a completely new pattern can be mined. At this time we would like to remind the reader that our library currently implements only four key kinds of patterns—sets, sequences, trees and graphs. Each of these marks a new strata of pattern complexity. For this example let us say we want to mine all frequent cliques, given an input dataset containing graphs. A clique of a graph is a maximal complete subgraph. Suppose we want to mine all frequent k-cliques, where k is the number of nodes in the clique. Since a clique is a specialized graph, we can guess that the process of mining cliques might resemble that of mining graphs. Let us reconsider the three core steps required for mining any patterns and compare the functionality in each of those for the two patterns. While the candidate generation step for graphs generates multiple candidates, the candidate generation step for cliques needs to generate only fully-connected graphs. This is much simpler than generating all possible candidates. The isomorphism checking and support counting for cliques does not change from regular graphs since cliques are specialized graphs. The alert reader might note that the task of mining cliques is similar to the task of mining

```
typedef proplist<directed,
        proplist<connected> > CLIQUE;

typedef proplist<directed > DI_GRAPH;

// Specialization for the clique pattern. //
template<class PAT, class MINE_PROPS,
         class SM_TYPE>
void
cand_gen(const pat_fam<proplist<directed,
                proplist<connected, PAT> >& Fk,
          ....);

// Specialization for directed graphs //
// Can be used by cliques.          //
template<typename T>
bool
check_isomorphism(pattern<proplist<directed,
                          T> >* cand_pat);
```

Figure 10: Adding a new pattern

itemsets. Although they are similar there is a subtle difference—itemsets are guaranteed to have unique labels whereas this is not the case with cliques. This argument reinforces our claim that cliques just differ in the isomorphism-checking step. Even though this example might seem contrived, it helps us see that a similar approach can be taken for any other pattern. In the worst case, the user will need to provide implementations for all three stages of pattern mining. From our experience with pattern mining, we can confidently claim that all the patterns in figure 6 along with many others need very few modifications on the part of the user. This has been the motivation behind the library design and implementation. Figure 10 shows the interface for the specialized candidate generation method for cliques. The first parameter is specialized to match a clique or any of its sub-concepts. The rest of the parameters have been omitted as they are not relevant to the example. Clique mining can borrow the remaining methods that are specialized for directed graphs.

6 Challenges and Future Work

The design and implementation of DMTL has helped us appreciate some of the language features provided by C++. While specialization by overloading, iterator categories, and similar powerful concepts are extremely important for generic programming, there are other aspects that are not equally well explored. Features such as *concept checking* and *named parameters* are features that would benefit our implementation. Moreover, dispatching based on concepts rather than pure type checking would allow partial specialization based on concepts. Even though some of these features have been implemented via template metaprogramming and made available in Boost libraries, our experience suggests advantages of including these features in the language standard.

The current design of DMTL has substantial scope for improvement. For example, our implementation of static lists to manage the pattern properties is not necessarily the best design choice. Such a property-list–based mechanism enforces a strict ordering of the properties in order for the compiler to select the appropriate specialization. Ideally, we would have benefited from the support for *named parameters* in C++. With such a feature we could omit the properties that did not apply for a specific pattern and provide property in any order. While *named parameters* seems like a good option, it might result in changes to the interface while introducing newer properties in our framework. A different approach to handling dispatching in this scenario would necessitate support for concept based dispatching as against type matching based dispatching. Additionally, support for concept checking [19] in the language specifications would enhance development efforts. We also explored using the **PropertyGraph** concept in BGL to represent a set of properties but it did not fit well into our framework at that point without compromising flexibility. The `enable_if` family of templates is an approach for enabling certain function templates and class template specialization. It could be used to achieving the same effect as our property list approach. We hope to explore this opportunity with other ongoing development in DMTL. From the data mining perspective, DMTL provides quite an extensive set of FPM algorithms which perform better than existing stand-alone algorithms. Since DMTL has been an evolving idea, now it is ready for its first public release after undergoing numerous refinements to the design. Some performance results based on an earlier version of DMTL are presented in our previous work [25]. In the long term, we plan to incorporate mining algorithms in other pattern spaces such as maximal patterns and closed patterns. Our eventual goal is to extend DMTL to other data mining tasks like classification, clustering, and so on.

7 Acknowledgment

We would like to thank Professor David Musser for his suggestions and feedback at various stages of this project.

8 References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *21st Int'l Conference on Very Large Data Bases*, 1994.

[2] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.

[3] Y. Chi, Y. Yang, and R. Muntz. Indexing and mining free trees. In *3rd IEEE International Conference on Data Mining*, 2003.

[4] T. E. Fawcett and F. Provost. Fraud detection. pages 726–731, 2002.

[5] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.

[6] B. Goethals. Frequent pattern mining implementations repository. `http://fimi.cs.helsinki.fi/`.

[7] T. Gschwind. PSTL—A C++ Persistent Standard Template Library. In *6th USENIX Conference on Object-Oriented Technologies and Systems*, 2001.

[8] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. Technical Report TR03-021, University of North Carolina, 2003.

[9] K. Knizhnik. Gigabase. `http://sourceforge.net/projects/gigabase`.

[10] R. Kohavi, D. Sommerfield, and J. Dougherty. Data mining using MLC++, a Machine Learning Library in C++. In *8th Int'l Conference on Tools with Artificial Intelligence*, 1996.

[11] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Second edition, 2001.

[12] S. Nijssen and J. Kok. Efficient discovery of frequent unordered trees. In *1st Int'l Workshop on Mining Graphs, Trees and Sequences*, 2003.

[13] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM/SIGMOD Int. Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 21–30, 2000.

[14] J. Siek, D. Gregor, R. Garcia, J. Willcock, J. Järvi, and A. Lumsdaine. Concepts for c++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2005. `http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2005/n1758.html`.

[15] J. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.

[16] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, 1995.

[17] B. Stroustrup and G. D. Reis. Concepts - design choices for template argument checking. Technical Report N1522=03-0105, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2003. `http://www.open-std.org/jtc1/sc22/WG21/docs/papers/2003/n1522.pdf`.

[18] J. Willcock, J. Järvi, A. Lumsdaine, and D. Musser. A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit, San Jose, CA*. Adobe Systems, Apr. 2004.

[19] J. Willcock, J. Siek, and A. Lumsdaine. Caramel: A concept representation system for generic programming. In *Second Workshop on C++ Template Programming*, Tampa, Florida, October 2001. `http://oonumerics.org/tmpw01/willcock.pdf`.

[20] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kauffman, 1999.

[21] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. Technical Report UIUCDCS-R-2002-2296, University of Illinois at Urbana-Champaign, 2002.

[22] M. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.

[23] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42:31–60, 2001.

[24] M. Zaki. Efficiently mining trees in a forest. In *8th ACM SIGKDD Int'l Conference on Knowledge Discovery and Data Mining*, 2002.

[25] M. Zaki, N. Parimi, N. De, F. Gao, B. Phoophakdee, J. Urban, V. Chaoji, M. Hasan, and S. Salem. Towards generic pattern mining. In *International Conference on Formal Concept Analysis (Invited Paper)*, 2005.

[26] M. J. Zaki, S. Jin, and C. Bystroff. Mining residue contacts in proteins. In *BIBE*, pages 168–175, 2000.

[27] L. Zhao and M. J. Zaki. Tricluster: An effective algorithm for mining coherent clusters in 3d microarray data. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 694–705, New York, NY, USA, 2005. ACM Press.

Changing Iterators with Confidence

A Case Study of Change Impact Analysis Applied to Conceptual Specifications

Marcin Zalewski and Sibylle Schupp
Dept. of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
{zalewski,schupp}@cs.chalmers.se

Abstract

Design and implementation of generic libraries in C++ are based on conceptual specifications—what if such specifications have to change? In a quite practical sense, this question arises because of a new proposal for iterator concepts that is under discussion among C++ library developers. Given the fundamental role of iterator concepts, it is important to anticipate which impact the proposed changes have on legacy code. Yet, no tool has been available to safely check for unwanted effects. We introduce a *conceptual* change impact analysis and apply it to the proposed iterator specification. Surprisingly, the analysis yields that the proposed iterator concepts are neither (fully) backward- nor forward-compatible with the current, standardized concepts. Since the analysis also lists the sources of incompatibility, it can help library designers to avoid unintended effects of their suggested changes and, in general, provides a base for assessing the impact of a conceptual change.

1 Introduction

In the design of today's generic libraries, so-called *iterator concepts* play a pivotal role in two ways. For one, they are *iterator* concepts, that is, abstractions of pointers that encompass operations for range traversal and data access at a granularity that makes them efficient basic building blocks of computations—in generic libraries, sequential sorting, graph traversal, matrix operations, or Fast Fourier Transforms, all are expressed in terms of iterator operations. Second, they are iterator *concepts* [11], that is, abstractions of types, which group syntactic, semantic, and behavioral requirements on types without being types themselves. Almost all interfaces of the parameterized components of a generic library are expressed in terms of concepts. Together, concepts, the hierarchy they typically form, and the conceptual interfaces they define, make out the conceptual specification of a generic library.

In C++, generic libraries use the hierarchy of iterator concepts that was introduced by the Standard Template Library (STL) [27] and became part of the language standard when STL was accepted (see [6, ch. 24]). Today, not only the algorithms of STL, MTL, BGL, Boost [1, 13, 14, 26], and of other generic libraries depend on these concepts, but also a large number of iterator *types* that were modeled after them as well as adaptors to those types and client libraries that instantiate iterator-based conceptual interfaces.

With the increased experience with STL iterator concepts, however, library developers started to encounter problems. As early as 2001, J. Siek pointed out that some generic algorithms are undergeneralized when expressed in terms of the currently available iterator concepts and submitted a new iterator specification for consideration by the C++ standard committee members [20]. Since then, his proposal has been subject of discussions and of a number of refinements [21–24]; the most recent, 5th revision (with additional authors) dates from April 2004. While this 5th proposal is not yet final, its continued discussion indicates a strong interest in revising the current, standardized iterator concepts.

Yet, the proposed changes are far from trivial: they include the introduction of new concepts, the omission and modification of old ones as well as modifications to the interfaces of the generic algorithms. It is therefore non-trivial to see whether any adverse effects on legacy code exist. It is also non-trivial to determine whether the changes to the iterator hierarchy have precisely the effects the authors intend them to have. Given the relevance of iterators, at the same time, and their ubiquity in generic libraries (in C++), it is of great importance that the impact of the introduced changes is well understood. Until now, however, their impact had to be determined by hand.

In this paper, we provide an automated assessment of the impact of the proposed changes to the iterator specification. Our assessment is based on *conceptual* change impact analysis (CCIA), i.e., change impact analysis applied to the conceptual specification of a library. CCIA is a general analysis technique for generic library maintenance that we currently develop. For this paper, we applied our prototype to two versions of the iterator hierarchy: the one defined in the working draft of the C++ standard [10] and the one submitted to the C++ committee by Siek, Abrahams, and Witt [24] (referred to from now on as *old* and *new* version, respectively). In our case study, we concentrate on the two kinds of intended impact that the new proposal sets out to make, namely to: (i) ensure backward- and forward-compatibility between old and new iterator hierarchy; (ii) reduce conceptual requirements on the parameters of STL algorithms, to increase their genericity.

To our surprise, the analysis shows that neither backward- nor forward-compatibility hold—as a consequence, algorithm requirements are not always reduced. At the same time, the analysis pinpoints the parts of the specification that break compatibility, thus can mark the first step towards aligning intended and actual effects. In some cases, it is the original concept definition that makes forward-compatibility hard to achieve, in other cases, the goals of the new proposal impede backward-compatibility. In yet other cases, compatibility problems come from the version we chose for the comparison—we use the iterator specification of the most recent (working) draft of the C++ standard, which was not available at the time of the new proposal.

Whether some, or all, of these incompatibilities are tolerable, is a

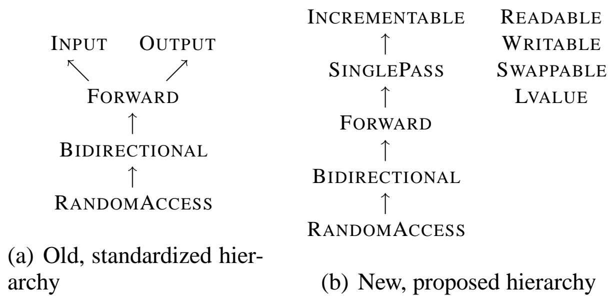
INPUTITERATOR	
operation	**type**
X u(a);	X
u = a;	X&
a == b	convertible to bool
a != b	convertible to bool
∗a	convertible to T
a→m	
++r	X&
(void) r++	
∗r++	convertible to T

SINGLEPASSITERATOR	
operation	**type**
++r	X&
r++	X
a == b	convertible to bool
a != b	convertible to bool

READABLEITERATOR	
operation	**type**
X u(a);	X
u = a;	X&
∗a	convertible to T
a→m	

T denotes the value type, *X* the type of the modeling iterator.

Figure 1. Concept tables of the INPUTITERATOR concept of the old hierarchy [10, Table 73] and its refactoring into the two concepts SINGLEPASSITERATOR and READABLEITERATOR of the new hierarchy [24]



(a) Old, standardized hierarchy

(b) New, proposed hierarchy

Figure 2. Old and new iterator hierarchies. The old hierarchy (a) is traversal-oriented, but mixes in value access. The new hierarchy separates the two concerns into a refinement hierarchy of traversal concepts (b.left) and 4 non-refining concepts for value access (b.right).

question the analysis does not seek to decide. The only purpose of any change impact analysis, not just our CCIA, is to flag an impact—whether this impact is wanted or unwanted, acceptable or unacceptable, is for the developers to judge. In the particular case of the iterator proposal, one has to weigh the benefits of increased genericity against the risk of breaking legacy code. Automatically generating a complete list of incompatibilities, our analysis can provide the base for such decision. Moreover, if the suggested changes lead to a revision of the currently standardized concepts, the detected incompatibilities can guide programmers in migrating to the new concept specifications.

While the core algorithm of our CCIA can deal with any conceptual changes, we focus in this paper on its application to the change of iterator concepts. We prepare the discussion of the case study with a description of the analysis proper, first informally in Section 3, then technically in Section 4. Section 5 details the set-up of the case study and Section 6 interprets the findings. To provide the necessary background, Section 2 summarizes the two iterator concept hierarchies under consideration. Sections 7–8 discuss related and future work, along with our conclusions.

2 Iterator Hierarchies

In generic libraries, algorithms are specified in terms of requirements on types, not in terms of types themselves. For these generic specifications, concepts are essential as they group requirements and allow expressing them in an abstract way. In specifying requirements abstractly, concepts might seem to resemble abstract classes or interfaces from object-oriented programming but there are two fundamental differences. For one, the requirements grouped by concepts comprise syntactic, semantics, and behavioral properties. Concept descriptions therefore contain not only signatures but also semantic constraints and complexity expressions. Second, and more importantly, concepts reside outside of type systems, to avoid imposing any requirements other than those of the concept description—in particular any implementation requirements, as they necessarily are established by abstract superclasses. Concepts and their formalization are an area of active investigation by a number of researchers (e.g., [25, 28]). Since we operate on an intermediate representation of concepts, however, our analysis is independent of a particular formalization.

For the discussion of the old and the new iterator concepts we use the following notation. A *concept* has one ("single-parameter concept") or more ("multi-parameter concept") parameters and groups *requirements* on its parameter(s). For the analysis, it suffices to consider syntactic requirements; Figure 1 contains a number of examples. There exist different kinds of relationships between concepts, requirements, and type parameters: A particular type (or types, for multi-parameter concepts) *models* a concept when it fulfills all requirements of the concept. A type parameter of a library interface is *constrained* by a concept if all actual parameter types have to model this concept. A concept, finally, can *refine* another concept where the *refining* concept includes all requirements of the *refined* concept; Figure 2 shows two refinement hierarchies. Refinement becomes more complicated when multi-parameter concepts are involved, because the parameters of the refining concept must be properly mapped to the ones of the refined concept.

Iterators are abstractions of range traversal and value access. In the old iterator hierarchy, each concept includes operations of both kinds. As the authors of the new iterator proposal point out, however, two problems arise when combining the concerns of range traversal and data access. On the one hand, some iterator types are intuitively incorrectly categorized with respect to their traversal protocol because of the additional value-access requirements they have to meet; for example the iterator type "vector⟨bool⟩ :: iterator" cannot model a RANDOMACCESSITERATOR concept, although the iterator types of all other vectors can (C++ Standard Library issue 96 [4] and Herb Sutter's paper [29]). On the other hand, some algorithm requirements are stricter than necessary, because value-access requirements cannot be separated from the requirements on range traversal. The new iterator concepts therefore are divided into two groups: traversal concepts on the one hand, value-access concepts on the other hand.

INPUT	
operation	**type**
∗a	T
++r	X&
(void) r++	
∗r++	T
a == b	bool

```
1 template ⟨class It1, class It2⟩
2 where {Input⟨It1⟩, Input⟨It2⟩}
3 bool equal (It1 first1, It1 last1, It2
     first2);
```

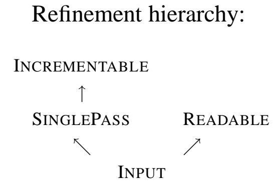
(a) Old specification

INPUT	
operation	**type**
—	—

INCREMENTABLE	
operation	**type**
++r	X&
r++	X

SINGLEPASS	
operation	**type**
a == b	bool

READABLE	
operation	**type**
∗a	T




```
1 template ⟨class It1, class It2⟩
2 where {Input⟨It1⟩, Incrementable⟨It2⟩, Readable⟨It2⟩}
3 bool equal (It1 first1, It1 last1, It2 first2);
```

(b) New specification

Figure 3. An example of a change in the conceptual specification of a library

New and old iterator hierarchies are depicted in Figure 2 (for brevity, the suffix ITERATOR has been omitted in all concept names); concept refinement is represented by the usual arrows. As the figure shows, the new hierarchy has five traversal concepts, corresponding to the traversal requirements that the old hierarchy defines; all value-access operations of the old hierarchy (except the index operator of RANDOMACCESSITERATOR) have been factored out into altogether four additional concepts. Every new concept, thus, contains a subset of the requirements of an old concept. Conversely, it is possible to reconstruct an original concept by re-defining it as the refinement of particular traversal and value-access concepts.

In illustration, Figure 1 shows the old INPUTITERATOR concept, which, in the new hierarchy, is split into two concepts, SINGLEPASSITERATOR and READABLEITERATOR; their combined requirements correspond to the original INPUTITERATOR requirements. The iterator proposal asserts that all new iterator concepts are backward-compatible with the corresponding old concepts and that all old concepts are forward-compatible with the corresponding new ones: "iterators that satisfy the old requirements also satisfy appropriate concepts in the new system" and "iterators modeling the new concepts will automatically satisfy the appropriate old requirements." We will return to the details of compatibility in Section 5. For the complete specification of the two hierarchies, we refer to their documentation [10, ch. 24], [24].

3 Example

Instead of plunging directly into the technical details, we introduce CCIA by means of an example. Given the original and a modified version of the conceptual specification of a library, we demonstrate how the impact of the changes is determined and how the analysis can be used to confirm, or question, (implicit) assumptions that underlie the changes.

Figure 3 shows the original and the modified conceptual specification of a simple, generic library loosely based on the iterator concepts introduced in the previous section. The original conceptual specification of the library consists of one concept and one algorithm, INPUT and "equal", respectively. The algorithm "equal" has two type parameters, "It1" and "It2", both constrained by the concept INPUT. In the new version of the specification three new concepts are added, the INPUT concept is modified, a refinement hierarchy is introduced, and the constraints on "It2" are rewritten. The new concept INCREMENTABLE contains two requirements related to range traversal. It is refined by the concept SINGLEPASS, which adds the ability to compare two iterators for equality. The third new concept, READABLE, consists of one requirement related to data access. For backward-compatibility, the INPUT concept is part of the new specification but it only refines the newly introduced concepts and does not have any requirements of its own. The constraints on the type parameter "It2", finally, are rewritten to INCREMENTABLE and READABLE.

As in the official iterator proposal, the intention behind the changes is to separate orthogonal concerns and to increase the granularity of concepts, yet without compromising compatibility. CCIA can help to automatically check whether the changes have the intended effect. In the particular context of our example, CCIA can help to check: (i) whether the INPUT concept represents the same set of requirements in the new and the old specification and (ii) whether the genericity of the algorithm "equal" increases by rewriting the constraints on "It2." CCIA cannot help to check whether the refactoring itself is correct and, for example, the orthogonal concerns have been factored out into the right concepts; questions of that kind, however, can hardly be automated. In the remainder of this section we describe the steps of the CCIA.

The analysis starts by encoding the original and the modified version of the specification. The concepts, requirements, and type parameters from Figure 3 and the relations between them are represented in the encoding. Next, the encodings of the two versions are compared and the difference between them is computed in a straightforward procedure. The comparison involves identifying entities and relations that exist in the old but not in the new version of the specification, and vice versa. As a result, for example, the concept INCREMENTABLE is marked as *added* since it exists in the new but not in the old version.

From the encoding, a directed dependence graph is constructed, which represents the two versions of the specification. The vertices correspond to concepts, requirements, and type parameters, while the edges represent the direction of change impact propagation implied by three kinds of relations: concepts including requirements, concepts refining concepts, and concepts constraining type parameters. A dependence graph for the specification in Figure 3 is shown in Figure 4.

The core algorithm of the analysis consists of two stages. First, the impact of the changes is propagated along the edges in the graph to identify the type parameters that may have been affected. Second, for every such potentially affected type parameter, the reversed (and appropriately reduced) dependence graph is traversed to find the sets of requirements that were added or deleted. For the example in Figure 3 the analysis detects that for the type parameter "It1" the

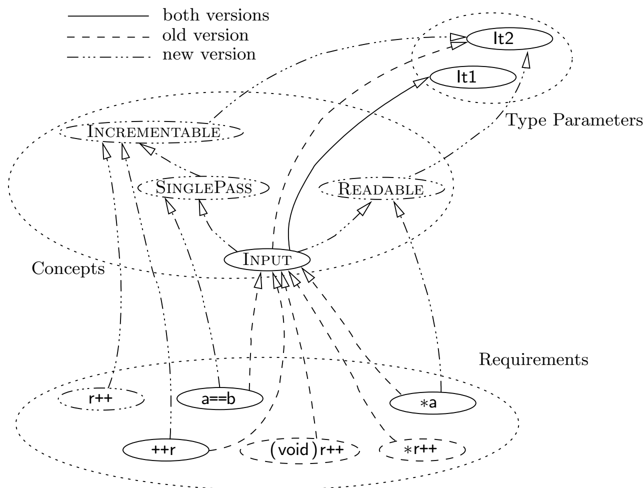


Figure 4. The dependence graph constructed for the specification from Figure 3

requirements "(void) r++" and "∗r++" were removed and "r++" was added and that for the parameter "It2" the requirements "(void) r++", "∗r++", and "a==b" were deleted and "r++" was added.

At this point the analysis is complete. What is left to do, is to relate the results back to the 2 original assumptions the library designer made, namely, that the concept INPUT still represents the same set of requirements as before and that rewriting the constraints on the parameter "It2" of the algorithm "equal" increases its genericity. As it is easy to see now, neither assumption is justified. For one, the requirements of the concept INPUT have changed. The change is implied by the change of requirements for the type parameter "It1", which is constrained by INPUT in both versions of the specification. Compared to the old specification, not only two requirements are deleted ("∗r++" and "(void) r++"), but also the requirement ("∗r++") is added. The new concept INPUT, therefore, is neither forward- nor backward-compatible with the old concept of the same name. Even if it would be compatible, the genericity of the algorithm "equal" would not have been increased. Although there are some requirements its parameter "It2" no longer has to meet ("(void) r++", "∗r++", and "a==b"), it has instead to meet an additional requirement ("r++"). Therefore, the genericity of the algorithm has not strictly increased.

4 Conceptual Change Impact Analysis

As the example in the previous section shows, CCIA is a two-pass procedure. The first pass propagates changes, thus is essentially a forward-reachability problem that determines what *may* have been impacted. The second pass depends on the particular change impact of interest, thus varies between different applications. In case of the new iterator hierarchy, where we would like to understand how the change impacts compatibility and the requirements on algorithm parameters, this second pass is a backward-reachability problem, combined with special filters. This section discusses in detail the 3 major parts of the CCIA that we informally introduced in Section 3: the representation of a conceptual specification and its change; the data structure to store relevant relations and changes, a dependence graph; and the analysis itself.

4.1 Intermediate Representation

Six constructs suffice to represent the conceptual specification of a library: 3 *entities* and 3 *relations*, directly corresponding to the relations that concepts establish (see Section 2):

- *Type Parameters*: Static parameters of the interfaces of a library
- *Concepts*: Sets of requirements
- *Requirements*: Operations, associated types, and any other properties required from actual parameters
- *Constrains-relations*: Relations between type parameters and concepts
- *Refines-relations*: Relations between concepts
- *Requires-relations*: Relations between concepts and requirements

For example, the concept SINGLEPASSITERATOR (see Figure 1) constitutes 4 requires-relations, defined by the 4 requirements listed in its concept table, and one refines-relation, to the concept INCREMENTABLEITERATOR (see Figure 2). The type parameter "It1" of the algorithm "equal" we discussed in Section 3 constitutes one constrains-relation to the INPUT iterator concept.

By definition, a change implies a "before" and "after." We therefore decided to encode the new and the old versions of a concept specification together and to express the changes as annotations. Any entity or relation that exists in the new version but not in the old one is marked as *added*, and any entity or relation that exists in the old version but not in the new one is marked as *deleted*. Since entities are "stand-alone" constructs, connected by relations only, a change in type parameters, concepts, or requirements does not affect any other parts of the conceptual specification unless it is propagated by requires-, refines-, or constrains-relations. To compute the impact of changes, it therefore suffices to focus on added or deleted relations.

In the current prototype, we perform the annotations as *added* and *deleted* manually; if concepts were first-class citizens, a compiler could easily perform the same task.

4.2 Dependence Graph

The six constructs representing a conceptual specification naturally map to vertices in a graph, where edges capture the dependencies between them. Using a dependence graph, we can formulate the algorithms of the analysis in terms of graph algorithms and can store intermediate results by updating the graph.

As suggested by the representation of change discussed in the previous subsection, the graph is constructed from both the new and the old version of the conceptual specification of a library. Instead of presenting the full algorithm for graph construction, we extend the graph from Figure 4 to show the details, for simplicity previously hidden. The extended graph is listed in Figure 5. As the figure shows, every entity, but also every relation is represented by a *main* vertex along with one or more parameter vertices; the only exception are type parameters, which are represented by a single vertex only since they, obviously, do not have parameters themselves. Main vertices are labeled with the entity or relation they represent, while their parameters are unlabeled. Since the example contains only concepts and requirements that depend on one param-

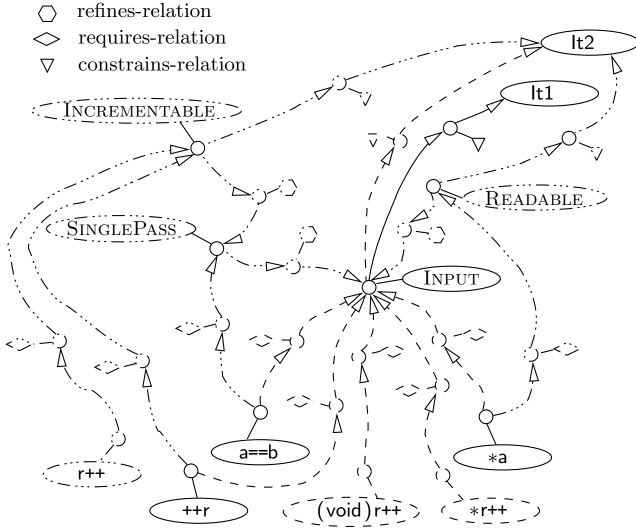


Figure 5. The graph from Figure 4, extended by previously hidden vertices

eter, all entities and relations have only one parameter vertex. The edges represent the direction of change impact propagation, i.e., invert the dependencies that exist in the conceptual specification.

The extended graph in Figure 5 consists of the same 4 concepts, 5 requirements, and 2 type parameters as the simple graph in Figure 4. As in the simple graph, a solid line indicates that a construct exists both in the old and the new specification, a dashed-and-dotted line that it is part of the new specification only, and a dashed line that it is part of the old specification only. For example, the INPUT concept exists in both versions of the specification, the requirement "r++" only in the new, and the requirement "(void)r++" only in the old specification. Type parameters are related to concepts through constrains-relations, concepts are related to concepts through refines-relations, and concepts are related to requirements through requires-relations. For example, the dashed edge from the parameter of the (single-parameter) concept INPUT through a constrains-relation to the "It2" type parameter means that any actual type that binds "It2" had to model the INPUT concept in the old specification. The dashed-and-dotted line from the parameter of INCREMENTABLE through a constrains-relation to "It2" means that any actual type that binds "It2" has to model the INCREMENTABLE concept in the new specification.

4.3 Algorithm Constraints Change

After the graph is constructed, the analysis performs two passes. In the first pass, the impact of changes is propagated: any vertex marked as added or deleted *may* have some impact on any of the vertices reachable from it. After the propagation of impact, the second pass of the analysis detects the actual effects of the changes. While the first pass always describes a forward-reachability problem, the second pass requires different algorithms, depending on the change impact investigated. In this case study, the second pass is based on backward-reachability but involves some additional logic.

The algorithm for propagating the impact of changes in pass 1 is rather straightforward. In short, it is a depth-first search where all vertices are found that are reachable from any vertex marked as added or deleted. The search stops on deleted vertices if the root of the current search path is an added vertex, and on added vertices if the root is a deleted vertex. The edges in the discovered paths are marked as change-propagating edges. The resulting modified dependence graph is used by the second pass of the analysis.

In this second pass we seek to answer whether the two concept hierarchies are compatible and whether the genericity of algorithms has increased. Both questions are addressed by the algorithm described below (see Algorithm 1: Constraints Change). Given an arbitrary type parameter and the dependence graph from pass 1, this algorithm finds all requirements that were added or deleted for that type parameter, i.e., all requirements implied by any of the constrains-, refines-, and requires-relations reaching that type. Since a constraint can be added or deleted multiple times, through different changes in the relations, the *Constraints Change* algorithm records with every change the path in the graph that leads to this change. A high-level definition of the algorithm follows:

Algorithm 1. Constraints Change.
Input: $\mathcal{G}$, a dependence graph where all change-propagating edges are marked; T, a type parameter vertex.
Output: R, a set of tuples (p,S) such that p is a path in $\mathcal{G}$ from T to a modified vertex q and S is a set of paths from q to the added or deleted requirements on T that result from the change in q.
Local: *reaching_paths*, a container of the results of [Find changes].
Notation and subroutines:

1. *path*: A path in a reversed dependence graph.
2. *forward or cross edge*: An edge (u,v) where v is colored black and not an ancestor of u in a search tree.
3. *last*(): Given a path, extracts the last vertex.
4. *significant_vertex*(): Given a parameter vertex, finds the main vertex of the corresponding relation or entity.

A1. [Filter and revert $\mathcal{G}$.] $\mathcal{G}'$ is $\mathcal{G}$ with all edges reversed and non-propagating edges removed.
A2. [Find changes.] Run depth-first search on $\mathcal{G}'$ with T as the root vertex:
A2.1. If a vertex marked as added or deleted is discovered, record current path in *reaching_paths*, mark vertex black, and backtrack depth-first search.
A2.2. If a forward or a cross edge is detected in the depth-first search, recursively run [Find changes] for the target of that edge. Merge all detected paths with the current path and record it in *reaching_paths*.
A3. [Process Changes.] For each path p in *reaching_paths* where s is *significant_vertex*(*last*(p)):
A3.1. If s is a constrains-relation, flatten the requirements of the constraining concept. Record the tuple (p, {*paths from the flattened concept to requirements*}) in R.
A3.2. If s is a refines-relation, flatten the requirements of the refined concept. Record the tuple (p, {*paths from the flattened concept to requirements*}) in R.
A3.3. If s is a requires-relation, record the tuple (p, {*one-vertex path of last*(p)}) in R.

We conclude this section by returning to the example from Section 3. Suppose we seek to validate the compatibility between the old INPUT concept and the corresponding concepts READABLE and SINGLEPASS of the new hierarchy. Before running the analysis, we first need to state the intended (forward-) compatibility by redefining the INPUT concept as a refinement of these two concepts, SINGLEPASS and READABLE. The analysis then starts by constructing the dependence graph of the 4 concepts involved (SINGLEPASS refines INCREMENTABLE), their requirements, the requires-relations,

and the refining-relations; the resulting graph, we have already seen in Figure 5. We currently need to distinguish by hand which entities and relation are old, and which ones are new. Once the graph is created, the *Constraints Change* algorithm is executed. *Constraints Change* then finds all requirements that have been added or deleted for each of the two parameters "It1" and "It2" of the algorithm "equal" (see Section 3 for the complete list). To decide whether compatibility holds, finally, the added and deleted requirements need to be compared. In the current prototype, we simply check whether every deleted requirement was added at least once and every added requirement was deleted at least once. Such simple comparison could find false positives, for example, if a deleted requirement continues to be associated with a type parameter through another, unchanged path in the graph. For this particular study, however, we are able to rule out those false positives: since all iterators are re-factored, no old and unchanged path remains. In the general case, false positives can be eliminated in a third, forward pass that checks for any added or deleted requirement whether other paths exist that neutralize the effect of addition or deletion, respectively. We have not yet implemented this pass, but the algorithm *Constraints Change* is prepared insofar it already associates changes and paths.

5 The Study

We now turn to the core of this paper, the case study, which applies CCIA to two versions of (conceptual) iterator specifications. In this comparison, the original iterator specification is taken from the C++ working draft [10], the new version from the iterator proposal submitted to the C++ standard committee [24].

Before the analysis can be conducted, the conceptual specifications have to be encoded in a form from which the dependence graph (see Section 4.2) can be constructed. Unfortunately, this encoding cannot be automated. Although large parts of an iterator specification in C++ are provided as a table of valid C++-expressions (see, e.g., Figure 1)—which in fact could be parsed and automatically processed—these tables are supplemented by auxiliary or qualifying definitions in natural language, sometimes given by means of examples. Further syntactic and semantic requirements are dispersed throughout the documentation—manual encoding is thus unavoidable. As one might expect, however, not all semi-formal descriptions map directly to a machine-usable format.

In this section we first specify our encoding scheme and explain for each kind of encoding how it is constructed. Next, we list the parts of the documentation we could not "naturally" express in our encoding scheme and make explicit the decisions we therefore had to make. We end the section with a description of the setup of the study and the format of the results.

5.1 Encoding Scheme

We encode the specification in terms of the entities and relations introduced in Section 4.1. An encoding of an entity or relation consists of a row of text, which itself is a triple of: a tag, corresponding to the encoded entity or relation (i.e., Type, Concept, Requirement, Constrains, Refines, Requires); the relation- or entity-specific information; and a flag (Added, Deleted, None) that indicates whether the entity or relation exists in the new but not old, old but not new, or in both specifications.

An excerpt of the encoding is shown in Figure 6. Line 1 indicates that the concept WRITABLEITERATOR was added in the new version and has two parameters: "Iter" and "Value." Line 2 means that the requirement "V o; I a; *a = o;", with parameters "I" and "V", exists in both versions, i.e., there exists a concept in the old, and a concept in the new version that both include this requirement. Line 3 associates this requirement with the newly added concept WRITABLEITERATOR and line 4 specifies that WRITABLEITERATOR refines COPYCONSTRUCTIBLE. Lines 5 through 8 show that the parameter of the algorithm "iter_swap", "ForwardIterator", was constrained by the FORWARDITERATOR concept in the old version and is constrained by the WRITABLEITERATOR and READABLEITERATOR concepts in the new version. The full encoding, of 330 lines, comprises both iterator specifications and the conceptual interfaces of STL algorithms. It is available on the accompanying web-page [2].

Although all encoded specifications are extracted from the documentation, the different kinds of encoding are based on different parts of the documentation and require different degrees of manual intervention. Concepts, to begin with, are relatively easy to encode. Since each table in the documentation corresponds to one concept, we can almost mechanically create one concept entity per table. We then only need to infer from the syntactic requirements in the concept table how many parameters the concept has: if the requirements refer to only one modeling type, the concept has one parameter only. All iterator concepts except the OUTPUTITERATOR concept in the old, and the WRITABLEITERATOR concept in the new specification are single-parameter concepts.

Encodings for requirement entities, next, are created one per row of the concept tables. Since the tables contain valid expressions in C++, we can use the *valid-expressions notation* that Stroustrup introduced [28]. This notation, very simply, specifies requirements in terms of ordinary C++ expressions. Using C++ as specification language makes minor adjustments in the interpretation of an expression necessary; the only deviation from the C++ semantics that is important in our context, however, concerns constructor expressions. In C++, the semantics of the expression "X a;" includes the declaration of a variable declaration and its default construction. In the valid-expression notation, the expression refers only to a variable declaration. Following the valid-expression notation, for example, the requirement "*a with the result convertible to T" (see the concept table of READABLEITERATOR in Figure 1) is encoded as "X a ; X::value_type v = *a;" (where, as just explained, the first expression denotes a declaration only, no default construction). Each requirement is recorded only once, i.e., if two concepts have a requirement that can be represented by the same abstract syntax, we encode this requirement only once, but add one requires-relation to each of the two concepts. Again, a concept-aware compiler would have already identified requirements that are identical at an abstract level.

While the encoding of requires-relations follows directly from the requirements we just discussed, the creation of refines-relations demands careful reading of the working draft of the C++ standard, since the refinement information is given at different places in the document. In the new iterator proposal, this task is made simple, as all refinement relations are always stated in the table headings.

The type parameter entities and their constrains-relations, finally, are encoded based on the section in the new iterator proposal that lists all changes to the conceptual interfaces of algorithms that result from the proposal. The changes are expressed as rewrite rules of the form "X $\rightarrow$ Y" where X and Y are type parameters that are constrained by concepts of the same name.

In illustration, Figure 7 lists the rewrite rule that applies to the second occurrence of the conceptual constraint INPUT in the two STL

```
1 Concept,"WritableIterator","Iter,Value",ADDED
2 Requirement,"V o; I a; *a = o;","I,V",NONE
3 Requires,"WritableIterator","Value o; Iter a; *a = o;","Iter,Value","I,V",ADDED
4 Refines,"WritableIterator","CopyConstructible","Iter","T",ADDED
5 Type,"iter_swap::ForwardIterator",NONE
6 Constrains,"ForwardIterator","iter_swap::ForwardIterator","Iter",DELETED
7 Constrains,"WritableIterator","iter_swap::ForwardIterator","Iter",ADDED
8 Constrains,"ReadableIterator","iter_swap::ForwardIterator","Iter",ADDED
```

Figure 6. An excerpt of the specification encoding, illustrating each of 6 encoding kinds

INPUT (2) $\longrightarrow$ INCREMENTABLE and READABLE
equal, mismatch

Figure 7. Example of a rewrite rule, applicable to (the second occurrence of) INPUT in the interface of the algorithms equal and mismatch. Parameters constrained by INPUT (2) are to be constrained by INCREMENTABLE and READABLE.

algorithms "equal" and "mismatch" and replaces there the (old) INPUT iterator concept by the two (new) concepts INCREMENTABLE and READABLE. Assuming that the corresponding concepts and type parameter have already been encoded, each rewrite rule therefore constitutes the encoding of at least 2 constrains-relations: one *deleted* relation to the constraining concept in the old version and *added* relations (1, or more if Y is a set) to the constraining concept in the new version. In the proposal, each rewrite rule is accompanied by the list of algorithms for which constraints should be rewritten.

Old concept	Corresponding new concepts
INPUT	READABLE, SINGLEPASS
OUTPUT	WRITABLE, INCREMENTABLE
FORWARD	READABLE, READABLELVALUE, FORWARD
MUTABLEFORWARD	READABLE, READABLELVALUE, FORWARD, BASICWRITABLE, WRITABLELVALUE
BIDIRECTIONAL	READABLE, READABLELVALUE, BIDIRECTIONAL
MUTABLEBIDIRECTIONAL	READABLE, READABLELVALUE, BIDIRECTIONAL, BASICWRITABLE, WRITABLELVALUE
RANDOMACCESS	READABLERANDOMACCESS, READABLE, READABLELVALUE, RANDOMACCESS
MUTABLERANDOMACCESS	READABLERANDOMACCESS, READABLE, READABLELVALUE, RANDOMACCESS, WRITABLERANDOMACCESS, WRITABLE, WRITABLELVALUE

Table 1. Correspondences between the old and the new iterator concepts

5.2 Design Decisions

The requirements of a conceptual specification often are expressed in a conditional form. For example, the return type of the valid expression "∗a" for FORWARDITERATOR is stated as "T if X is mutable, otherwise const T& " (where *T* denotes the value type, *X* the type of the modeling iterator) [10, Table 75]. Other conditional specifications in disguise have the form of optional (type) qualification or different return types of overloaded expressions. Since in our scheme, conditional requirements cannot be expressed in a straightforward way, we had to modify the conceptual hierarchies by introducing new concepts that represent alternative branches of conditions.

In the old hierarchy, we introduced the 3 concepts MUTABLEFORWARDITERATOR, MUTABLEBIDIRECTIONALITERATOR, and MUTABLERANDOMACCESSITERATOR, corresponding to the "if mutable"-condition in the specification. In the new hierarchy, we replaced the concept LVALUEITERATOR by the two concepts READABLELVALUEITERATOR and WRITABLELVALUEITERATOR, to capture the optional *cv*-qualification of the return type of the dereferencing operator "∗." Moreover, we added the concepts READABLERANDOMACCESSITERATOR and WRITABLERANDOMACCESSITERATOR as refinements of RANDOMACCESSITERATOR, to match the precondition "pre: a is a readable iterator" of the "a[n]" operation and the precondition "pre: a is a writable iterator" of the "a[n]=v" operation, respectively [10, Table 77]. For a different reason, finally, we introduced the concepts BASICOUTPUTITERATOR and BASICWRITABLEITERATOR. They are single-parameter variants of the concepts OUTPUTITERATOR and WRITABLEITERATOR, which have two parameters, thus cannot be directly refined by any single-parameter iterator concept.

We also had to make a decision whether or not FORWARDITERATOR and MUTABLEFORWARDITERATOR refine both INPUTITERATOR and OUTPUTITERATOR. This relation is not clear, since the C++ standard, on the one hand, states in the introductory paragraphs to the iterator specification that the FORWARDITERATOR concept includes the requirements of the INPUTITERATOR and OUTPUTITERATOR concepts. On the other hand, the tabular specification of the FORWARDITERATOR concept contains requirements that conflict with this statement (see library issue 299 on the C++ standard web-page [3]). After careful consideration we have decided to include the refinement in our specification as it seems to reflect the common understanding of the iterator concepts.

Table 1 lists all concepts that are included in this case study along with their correspondence relation. Using this table, compatibility can be decided row-wise: a concept in the old hierarchy is forward-compatible if any type modeling the concept also models the new concepts in the same row. Conversely, a concept of the new hierarchy is backward-compatible if every modeling type also models the old concept in the same row.

5.3 Setup

The setup of the case study is now easy to explain. To check the compatibility between the old and the new concepts, we proceed essentially as already illustrated in Section 4.3: based on the expected compatibilities defined in Table 1, we redefine all old concepts in terms of their counterparts in the new proposal, i.e., we mark all requires- and refines-relations from the old specification as *deleted* and then *add* refines-relations from every old concept to the corresponding new one(s). Next, we create 9 type parameters—one per concept parameter of the old hierarchy (recall that there are 8 concepts in the old hierarchy and OUTPUTITERATOR has 2 parameters)—and *add* and *delete* constrains-relations that reflect their changed constraints from the old concept to the new corresponding concepts. Then, we call the routine for constructing the dependence graph and apply the algorithm *Constraint Change* to each type parameter we have added. If any requirements are

	Con.	Requirement	
1	**O**	typename Iter::value_type;	b
2	**O**	Iter r; Iter q = r++;	f
3	**O**	typename Iter::difference_type;	b
4	**O**	typename Iter::pointer;	b
5	**O**	typename Iter::reference;	b
6	**O**	Iter r; const Iter& q = r++;	b
7	**O**	Iter r; V o; *r++ = o;	~~b~~
8	**I**	Iter r; Iter q = r++;	f
9	**I**	typename Iter::difference_type;	b
10	**I**	typename Iter::pointer;	b
11	**I**	typename Iter::reference;	b
12	**I**	Iter r; r++;	~~b~~
13	**I**	Iter r; Iter::value_type q = *r++;	~~b~~
14	**F**	Iter r; const Iter::value_type& q = *r++;	~~b~~
15	**MF**	Iter r; Iter::value_type& o = *r;	f
16	**MF**	Iter r; Iter::value_type o; *r++ = o;	~~b~~
17	**B**	Iter r; Iter::value_type q = *r−−;	~~b~~
18	**RA**	Iter r; Iter::value_type q = r[n];	f
19	**RA**	Iter r; const Iter::value_type& q = r[n];	b
20	**MRA**	Iter r; Iter::value_type v; r[n] = v;	f

where **O**=OUTPUT, **I**=INPUT, **F**=FORWARD, **MF**=MUTABLEFORWARD, **B**=BIDIRECTIONAL, **RA**=RANDOMACCESS, **MRA**=MUTABLERANDOMACCESS

Table 2. The requirements that cause forward-incompatibility (f) or backward-incompatibility (b). False positives are indicated by ~~stricken~~ text.

deleted but not added, backward-compatibility is broken. Correspondingly, if any requirements are added but not deleted, forward-compatibility is broken.

To calculate the changes in the requirements of STL algorithms, we encode the rewrite rules from the iterator proposal (see Section 5.1) and apply Algorithm 1, *Constraint Change*, to the type parameters of the algorithms. The genericity of an algorithm is increased if there are requirements that were deleted but not added, and no other requirements were added but not deleted.

6 Results

The analysis returns its output in different formats, which can be controlled by the user through flags. Three of these output formats are related to the issue of compatibility: with increasing verbosity, *compatibility summary* summarizes for each concept whether or not it is compatible, *compatibility short output* lists how many times a requirement was deleted or added, and *compatibility incompatible* shows the requirements causing the incompatibility of a particular concept. A fourth output format, *genericity change*, summarizes for all type parameters of all algorithms whether or not their genericity was increased. The full and unprocessed results of Algorithm 1 can also be turned on. Figure 8 shows examples of the output for each kind of format; the complete traces are available on the webpage accompanying this paper [2]. In this section, we present and interpret the results of the case study.

6.1 Forward- and Backward-Compatibility

Surprisingly to us, the analysis yields that new and old iterators are not compatible. More specifically, none of the 8 old concepts and their corresponding new concepts (in the sense of Table 1) is backward- or forward-compatible. Even if we ignore incompatibilities propagated through the refinement hierarchy, there are only 3 concepts that introduce no incompatibilities on their own: FORWARDITERATOR and BIDIRECTIONALITERATOR (yet, see the discussion below), and the MUTABLEBIDIRECTIONALITERATOR concept that we had to introduce (Section 5.2). These 3 concepts will be automatically both backward-compatible and forward-compatible if their refined concepts are "fixed."

Table 2 details the incompatibilities. Following the refinement hierarchy, the table lists for each concept exactly the incompatibilities this concept introduces, i.e., omits those incompatibilities that are only propagated through refinement. Each row, thus, corresponds to one incompatibility; the kind of incompatibility is indicated in the last column. For example, line 1 of the table indicates that the associated type "value_type" of the old specification of OUTPUTITERATOR is missing in the specification of the corresponding new concepts (WRITABLEITERATOR and INCREMENTABLEITERATOR), which breaks backward-compatibility. A further 6 incompatibilities of the OUTPUTITERATOR concept are given in lines 2-7. They all propagate to all refining concepts—that is, all other concepts except the INPUTITERATOR concept—but are not listed again in the table.

It is important to note that some incompatibilities detected by our analysis are in fact wrong, albeit in a subtle way that shows a general limitation of our approach. We indicate all faults on part of the analysis by stricken text in the last column of Table 2. As the table shows, there are 6 cases of false positives: 5 due to the compound expressions "*r++" and "*r−−" (lines 7,13,14,16, and 17) and one (line 12) caused by the "r++" expression. The latter is a false positive because it is implied by the requirement "Iter r; Iter q = r++;" of the INCREMENTABLEITERATOR concept. The former ones, coming from compound expressions, are due to the granularity of our analysis, which looks at expressions in an atomic way and therefore compares only expressions for equality, not their compositions. Thus, if a compound expression is changed into its constituents, the analysis only recognizes that the compound expression is deleted and certain new expressions are added, but does not attempt to determine whether a composition of expressions exist that is equivalent to the deleted compound one. Exactly such decomposition, however, takes place in the iterator proposal: the requirement "* r++" of the old proposal, which merges the concerns of traversal and value access, is decomposed into two requirements, "r++" and "*r" (which are then associated to different concepts). For some concepts, for example the FORWARDITERATOR concept, the corresponding new concepts define these 2 new valid expressions so that their composition in fact is identical to the original compound expression. Not knowing of this identity, however, our analysis flags these expressions as backward-incompatible.

For the particular cases of the iterator proposal, it would be quite easy to establish the identity of "*r++" and its two constituents in an ad-hoc fashion. For a systematic handling of implied identities, however, the analysis would have to be extended by an extra inference step. A simpler alternative might seem to avoid these false positives altogether, by breaking composite requirements down into their constituents and representing them in the dependence graph as a sequence of non-compound expressions. Yet, such decomposition changes the semantics of a requirement since the sequential execution implies the existence of a temporary, which, for example, cannot be assumed for the INPUT iterator concept.

The incompatibilities in Table 2 can be grouped into 3 categories. The most interesting ones are the ones that come from the separation of traversal and access concerns—the main motivation of the new proposal. In the old iterator concepts, these concerns were combined not just in the concept specification as a whole, but some-

Full:
```
InputIteratorModel --> constrains --> Iter (of InputIterator) --> refines -->  (DELETED)
     T (of CopyConstructible) --> requires --> T (of T t; T(t);)
     T (of CopyConstructible) --> requires --> T (of const T u; T(u);)
InputIteratorModel --> constrains --> Iter (of InputIterator) --> refines -->  (ADDED)
     Iter (of ReadableIterator) --> requires --> T (of typename T::value_type;)
     Iter (of ReadableIterator) --> requires --> T (of T::value_type v; T p; v = *p;)
```
Compatibility-summary:
```
InputIteratorModel NOT COMPATIBLE
ForwardIteratorModel NOT COMPATIBLE
```
Compatibility-incompatible:
```
InputIteratorModel
     Requirement "T t; T q = t++;" added 1 times, deleted 0 times. (FORWARD INCOMPATIBLE)
     Requirement "typename T::difference_type;" added 0 times, deleted 1 times. (BACKWARD INCOMPATIBLE)
```
Compatibility-short:
```
OutputIteratorModelIter
     Requirement "T t; T u; T& q = (t = u);" added 1 times, deleted 1 times.
     Requirement "T t; const T v; T& q = (t = v);" added 1 times, deleted 1 times.
```
Genericity-change:
```
find_first_of::ForwardIterator2  --- Genericity not increased.
```
Figure 8. Examples of five different kinds of output from the analysis

algorithms	Del.
reverse_copy, find_end, adjacent_find, search, search_n, rotate_copy, lower_bound, upper_bound, equal_range, binary_search, min_element, max_element	1
find_first_of	3, 4
copy_backwards	1, 0
equal, mismatch, transform	4

Table 3. STL algorithms with increased genericity, grouped by the number of requirements removed per parameter (second column); backward-compatibility is provided.

times in one requirement; we have already seen the example "∗r++." Separating these expressions in a traversal-expression on the one hand, a value-access expression on the other hand, the new hierarchy cannot always define them so that the original compound expression remains valid. Lines 2, 6, and 8 in Table 2 show such incompatibilities.

A second group of incompatibilities are associated types; generally, the new concepts have fewer associated types than the old ones (see lines 1,3-5,9-11). This difference is a result of the proposal only requiring the minimal set of associated types from the new concepts. For example, while every old concept is required to have four associated types ([10, Sec. 24.3.1]), it makes no sense to require the OUTPUTITERATOR concept that does not provide a difference operation to define "difference_type".

The final kind of incompatibilities results from the intended compatibility with the latest C++ standard. In our study, we used the more recent draft version of the standard, because it corrects some mistakes and resolves some ambiguities in the natural-language specification of iterators. Compared to the standard, however, the draft also changes the return type of the index operator "r[n]", resulting in the two incompatibilities listed in lines 18 and 19. In addition, the incompatibility in line 15 is caused by different return type of the dereference operator "∗a" of the FORWARDITERATOR concept in the draft and in the standard. The last expression in line 20, finally, "Iter r; Iter ::value_type v; r[n] = v;", is not required by the old concepts, neither in the draft and nor in the actual standard, but was added in the new proposal to fix a problem in the old iterator hierarchy (C++ issue 299 [3]).

6.2 Algorithm Requirements

Refactoring the iterator hierarchy would be an academic exercise if it would not allow rewriting the constraints on STL algorithms so that they become more generic. In fact, the main goal of the proposal, as pointed out in the introduction, was to be able to increase the genericity of STL and other iterator-based libraries. The proposal therefore includes a set of rewrite rules that define for each STL algorithm how the constraints on its parameters can be rewritten in the presence of the new concepts. One example of such rewrite rule we have already seen in Figure 7. There, the underlying assumption was that rewriting INPUT as INCREMENTABLE and READABLE relaxes the constraints on certain parameters of the algorithms "equal" and "mismatch". The backward-incompatibilities, however, that we reported in the previous section (Table 3), inevitably invalidate any such assumptions since every backward-incompatibility introduces an additional requirement, which the parameters originally did not have to meet. For example, changing constraints as suggested in Figure 7 introduces for the algorithms "equal" and "mismatch" the additional requirement "Iter r; Iter q = r++;" (line 12, Table 3), which comes from the INCREMENTABLE concept but was not included in the old INPUT concept. It follows that it impossible for the analysis to confirm that the genericity of any algorithm has been increased.

Since it is of interest nevertheless to assess how much the genericity could increase we conducted an experiment where we bypassed all incompatibility issues. Assuming, to that end, that all compatibilities hold as intended and defined in Table 1, we did not directly apply the rewrite rules that the proposal specifies. Instead, we expressed these rewrite rules in terms of new concepts only: using the intended correspondences, we replaced old concepts on the right-hand sides of the rewrite rules by their corresponding new concepts; Figure 9 shows how the rule from Figure 7 is transformed. The "adjusted" rules represent the change in the genericity *intended* by the proposal authors by neutralizing the unwanted effects of incompatibilities (the new concepts are compatible with themselves) but preserving the effects of the original rules on the genericity of algorithms. We applied CCIA to two versions of the STL specifications that both use the new concept hierarchy but differ in the constraints on algorithm parameters as prescribed by our "adjusted" rewrite rules. Although the transformation of the original rules is ad-hoc, it allows us to determine which algorithms in STL benefit from the new iterator concepts.

READABLE and SINGLEPASS → INCREMENTABLE and READABLE
equal, mismatch

Figure 9. Modified rewrite rule from Figure 7. The old concept INPUT is replaced by its corresponding new concepts, see Table 1.

Table 3 shows for which algorithms their genericity increases provided the intended compatibility between the new and the old iterators holds. The algorithms are grouped by the number of requirements removed for each of their type parameters. From the 42 STL algorithms that are affected by the changes to the iterator concepts, 17 became more generic.

7 Related Work

Change impact analysis (CIA) describes no particular technique, but rather a collection of techniques varying with the purpose of the analysis. To convey an impression of the potential breadth of analyses, we refer to the recent *Guidelines for the Oversight of Software Change Impact Analysis Used to Classify Software Changes as Major or Minor* by the US Federal Aviation Administration (FAA) [5]. Somewhat to the extreme, these guidelines suggest as many as 10 analyses: traceability, memory and timing margin, data and control flow, input/output, development environment, operational characteristics, certification maintenance, and partitioning analysis. Bohner and Arnold [8] provide an overview of the most frequently used analyses, traceability and dependency.

In applications to software evolution or early phases of the software life cycle, CIA essentially requires the identification or classification of the computed effects. In applications to later phases of the software cycle, the identification of change impact often marks the first step only, since these effects must be communicated to other tools or analyses. Our CCIA falls in the first category as it is typical for CIAs based on specifications or requirements documents (e.g., [7, 15, 30]). In the latter category, in particular the number of applications to regression testing stands out (e.g., [16, 18]).

The complexity of CIA justifies its use in large software systems when changes are difficult to detect since their effects are non-local. Non-locality of impact in imperative programming usually comes from side effects. In object-oriented programming (OOP), subclassing, dynamic binding, and polymorphism are sources of non-locality. Dating back at least to Kung et al. [12], much work has been done to provide CIAs for OOP at different (e.g., method or class) levels of granularity and for different purposes. Since our analysis applies to generic libraries, where the non-locality of impact is due to the separation of concepts and types, its closest counterparts are class-level CIAs as for example the one by Rajlich [17], although we do not use his snapshot model for change propagation. Inspired by the notion of "atomic changes" [19], our representation of change implies that all changes take place simultaneously. The dependence graph itself is similar to the program dependence graphs of Horwitz et al. [9], even though their graphs represent dependencies between components of an imperative program while we represent dependencies in library specifications. Of the three classes of problems they make out as applications of dependence graphs, i.e., slicing, differencing, and integration, our analysis falls into the class of differencing problems.

CIA is often implemented using program slicing [31]. Since our CCIA concerns conceptual specifications, the classical slicing criterion (s, v), where s is a statement and v a variable used in s, is not applicable. Yet, if we allow as a slicing criterion a single entity or relation and base slicing not on a call graph, but on a graph of the conceptual specification, then the two passes of the CCIA can be understood as forward- and backward-slices, respectively, on a type parameter through the dependence graph.

8 Conclusions and Future Work

Among designers of generic libraries in C++, there is an ongoing discussion about changing the specification of iterator concepts—the basis of STL and many other generic libraries. Because of the fundamental role of iterator concepts, the effects of the proposed changes have to be well-understood. Especially important are the compatibility between the old and the new iterator concepts and the impact of the new concepts on the genericity of (legacy) libraries. Thus far, however, no automated tools were available that could determine the impact of conceptual changes. We have introduced a conceptual change impact analysis (CCIA) and applied it to the standardized and the proposed versions of iterator concepts. The analysis shows that the two iterator hierarchies are neither forward- nor backward-compatible and lists the parts of the specification that cause incompatibility. Its results can help library designers to avoid unintended effects of a change and, in general, provides a base for assessing its impact.

At present, the CCIA is still a prototype. Our plans for the immediate future include increasing the accuracy of the analysis by adding a third pass, which detects whether the deletion or addition of a requirement in fact introduces or eliminates the requirement *completely* or just changes its multiplicity (i.e., the number of ways in which it reaches a type parameter). The additional pass is needed when changes are small, e.g., in incremental concept development. It was not necessary in the study of iterator concepts, where the changes are so extensive that no relation remains unchanged.

In the more distant future, we want to investigate how the genericity of the analysis itself can be increased. Its usability could be significantly improved if we could identify (abstract) primitives that underlie conceptual change impact analysis. Instead of hard-wiring the investigation of compatibility, as we currently do, we could organize these primitives as basic building blocks and allow users to combine them according to the change impact of their interest. Yet, we need to gain much more experience with conceptual change impact, before we can try identifying such primitives.

At the implementation level, we hope to integrate the analysis with a C++ compiler, to automate the process of reading in conceptual specifications. There is work underway elsewhere to support C++ programs that are extended by concepts (e.g., [25, 28]). Such support provided, our users can be spared the tedious and error-prone manual encoding that is currently necessary.

Acknowledgments

We thank the anonymous reviewers for their thorough comments and suggestions, which were essential for improving the presentation of this paper. We also thank the authors of the Boost Graph Library (BGL), which saved us substantial implementation work. Douglas Gregor provided input for the necessary mapping from the semi-formal specifications in the C++ standard to a machine-usable format.

9 References

[1] *The Boost initiative for free peer-reviewed portable C++ source libraries*, http://www.boost.org.

[2] *Conceptual change impact analysis*, http://sms.cs.chalmers.se/index.php?title=CCIA.

[3] *C++ Standard Library issue 299: Incorrect return types for iterator dereference*, http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#299.

[4] *C++ Standard Library issue 96: Vector<bool> is not a container*, http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#96.

[5] Federal Aviation Administration, *N8110-85. Guidelines for the oversight of software change impact analyses used to classify software changes as major or minor*.

[6] ANSI-ISO-IEC, *C++ standard, ISO/IEC 14882:2003(E)*, ANSI standards for information technology ed., October 2003.

[7] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, *Recovering traceability links between code and documentation*, IEEE Trans. Softw. Eng. **28** (2002), no. 10, 970–983.

[8] S. A. Bohner and R. S. Arnold, *Software change impact analysis*, Wiley-IEEE, 1996.

[9] S. Horwitz and T. Reps, *The use of program dependence graphs in software engineering*, Proc. of the 14th Internat. Conf. on Softw. Eng., 1992, pp. 392–411.

[10] ISO/IEC JTC1/SC22/WG21 - C++, *C++ standard draft, n1804=05-0064*, ANSI standards for information technology ed., October 2003.

[11] D. Kapur, D.Musser, and A.Stepanov, *Tecton: A language for manipulating generic objects*, Proc. of a Workshop on Progr. Specification, LNCS, vol. 134, Springer, 1981, pp. 402–414.

[12] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, *Change impact identification in object oriented software maintenance*, Proc. of the Internat. Conf. on Softw. Maintenance (ICSM), 1994, pp. 202–211.

[13] L. Lee, J. Siek, and A. Lumsdaine, *The generic graph component library*, Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages & Applications, vol. 34, 1999, pp. 399–414.

[14] D. Musser, G. Derge, and A. Saini, *STL tutorial and reference guide. C++ programming with the Standard Template Library*, 2nd ed., Addison Wesley, 2001.

[15] J. O'Neal, *Analyzing the impact of changing requirements*, Proc. of the Internat. Conf. on Softw. Maintenance (ICSM), 2001, pp. 190–198.

[16] A. Orso, T. Apiwattanapong, and M. Harrold, *Leveraging field data for impact analysis and regression testing*, Proc. of the 9th Europ. Softw. Eng. Conf. & 11th ACM SIGSOFT Internat. Symp. Foundations of Softw. Eng., 2003, pp. 128–137.

[17] V. Rajlich, *A model for change propagation based on graph rewriting*, Proc. of the Internat. Conf. on Softw. Maintenance (ICSM), 1997, pp. 84–91.

[18] G. Rothermel and M. J. Harrold, *A safe, efficient regression test selection technique*, ACM Trans. on Softw. Eng. and Methodology **6** (1997), no. 2, 173–210.

[19] B. Ryder and F. Tip, *Change impact analysis for object-oriented programs*, Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2001, pp. 46–53.

[20] J. Siek, *Improved iterator categories and requirements*, Tech. Report J16/01-0011 = WG21 N1297, ISO/IEC JTC1/SC22/WG21 - C++, March 2001.

[21] J. Siek, D. Abrahams, and T. Witt, *New iterator concepts*, Tech. Report N1477=03-0060, ISO/IEC JTC1/SC22/WG21 - C++, April 2003.

[22] ———, *New iterator concepts*, Tech. Report N1531=03-0114, ISO/IEC JTC1/SC22/WG21 - C++, September 2003.

[23] ———, *New iterator concepts*, Tech. Report N1550=03-0133, ISO/IEC JTC1/SC22/WG21 - C++, October 2003.

[24] ———, *New iterator concepts*, Tech. Report N1640=04-0080, ISO/IEC JTC1/SC22/WG21 - C++, April 2004.

[25] J. Siek, D.Gregor, R. Garcia, J. Willcock, J. Järvi, and A. Lumsdaine, *Concepts for C++ 0x*, Tech. Report N1758=05-0018, ISO/IEC JTC1/SC22/WG21 - C++, January 2005.

[26] J. Siek and A. Lumsdaine, *The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra*, Internat. Symp. on Computing in Object-Oriented Parallel Environments, 1998, pp. 59–70.

[27] A. Stepanov and M. Lee, *The Standard Template Library*, Tech. Report HPL-95-11, Hewlett Packard, November 1995.

[28] B. Stroustrup and G. Dos Reis, *A concept design (rev.1)*, Tech. Report N1782=05-0042(rev.1), ISO/IEC JTC1/SC22/WG21 - C++, April 2005.

[29] H. Sutter, *vector⟨bool⟩ is nonconforming, and forces optimization choice*, Tech. Report J16/99-0008 = WG21 N1185, ISO/IEC JTC1/SC22/WG21 - C++, February 1999.

[30] R. Turver and M. Munro, *An early impact analysis technique for software maintenance*, Journal of Software Maintenance **6** (1994), no. 1, 35–52.

[31] M. Weiser, *Program slicing*, IEEE Transactions on Software Engineering **10** (1984), 352–357.

Metadata-Driven Library Design

Antonio Cisternino
Dipartimento di Informatica, Università di Pisa,
L.go Bruno Pontecorvo 3, I-56127 Pisa, Italy,
cisterni@di.unipi.it

Walter Cazzola
Department of Informatics and Communication,
Università degli Studi di Milano,
Via Comelico 39/41, Milano, Italy.
cazzola@dico.unimi.it

Diego Colombo
IMT - Institutions, Markets, Technologies
Lucca Institute for Advanced Studies
Via San Micheletto, 3
55100 Lucca, Italy

Abstract

Library development has greatly benefited by the wide adoption of virtual machines like Java and Microsoft .NET. Reflection services and first class dynamic loading have contributed to this trend. Microsoft introduced the notion of custom annotation, which is a way for the programmer to define custom meta-data stored along reflection meta-data within the executable file. Recently also Java has introduced an equivalent notion into the virtual machine. Custom annotations allow the programmer to give hints to libraries about his intention without having to introduce semantics dependencies within the program; on the other hand these annotations are read at run-time introducing a certain amount of overhead. The aim of this paper is to investigate the impact of this new feature on library design, focusing both on expressivity and performance issues.

1 Introduction

Reflection and dynamic loading are becoming essential elements of modern programs. Their usefulness is testified, for example, by the JDBC architecture that shows how to implement a driver based architecture exploiting the Java dynamic loading.

Although reflection can be used to inspect the structure of types, to access fields and even to invoke methods dynamically, the concept of tagging has been anticipated as an interesting application. Consider for instance the Java serialization architecture: the programmer can declare the instances of a serializable class simply by implementing the `Serializable` interface, which in fact is an empty interface. Thus two types that differ only for the implementation of the `Serializable` interface are indistinguishable from the execution standpoint. Besides, the serialization of the instances of non-serializable types will not be allowed by the serialization support. Java serialization taught us that the meta-data stored with the code can be used for other purposes than mere execution. Other programs may rely on the reflective abilities of inspecting the compiled types and act differently depending on what they have found.

Although widely used by Java programs, the idea of providing explicit meta-data support for annotation has been introduced by Microsoft in the Common Language Runtime (CLR). The virtual execution environment is part of the CLI standard [Mil03][ECM]. More recently also Java introduced annotations as a mean of storing custom data inside Java classes [Java]. There are also proposals to add extensible reflection to C++ language [AC02].

Custom annotations have shown to be useful because they provide a channel that library-users and library-developers may use to communicate. A library may require that the user puts annotations on top of classes and methods in order to instruct the library on how to use it.

Unfortunately the availability of this new mechanism increases the number of possible choices a library developer has for modeling the abstractions to be provided to the final user of its library. The choice of using custom annotations instead of more traditional programming abstractions should be subjected to consideration about expressiveness and performance issues.

The paper is organized as follows: section 2 introduces custom annotations; section 3 is devoted to discuss how annotations have been used so far in real applications; performance considerations are presented in section 4; section 5 presents conclusions. As a final remark, throughout the rest of this paper we will also refer *custom annotations* as custom attributes and we will use the C# notation inside the examples.

2 Custom Attributes

A *custom attribute* is a piece of information attached by the programmer to a portion of a program. In the model implemented both in Java and .NET attributes can be attached only to those elements accessible through the reflection API, such as assemblies, types (delegates, value types, and classes), fields, properties, and methods; however there has been a proposal of extending the annotation model to code blocks in [AC02].

In .NET custom attributes are represented by instances of classes that inherit from the system class `Attribute`. Java exposes annotations as instances of an interface.

A custom attribute is defined by specifying a set of values and the type of the attribute; all the values used to create it must be computable at compile time. The following is an example of annotations in C#:

```
[MyAnnotation("par", Property="val")]
public class MyClass {..}
```

The definition of MyAnnotation attribute can be the following:

```
class MyAnnotationAttribute : Attribute {
   MyAnnotationAttribute(string par) {...}
   public string Property;
}
```

Parameters required to instantiate custom annotations are stored inside the binary file, along with the rest of reflection meta-data, so that they can be retrieved at run-time. This data is *ignored* by the execution environment unless explicitly accessed through the reflection API. For instance, let m be an instance of `MethodInfo` class (a reflective descriptor of a method), in C# we can retrieve the custom attributes associated with the method as follows:

```
Attribute[] attr = m.GetCustomAttributes();
```

The crucial idea behind the custom annotation consists of shifting up data about the code into the executable and to be available at run-time. Custom annotations are interpreted by programs and are used for program transformation.

A stereotypical example, from Microsoft .NET, of custom attributes usage is the support for implementing web services by means of custom attributes. `WebMethod` attribute is used to label methods that should be exposed as web services. A minimal web service written in C# that computes the sum of two integers is the following:

```
public class HelloWorldWS {
  [WebMethod]
  public int add(int i, int j) {return i+j;}
}
```

Once compiled, the `HelloWorldWS` class does not provide any web services interface. A different program - actually part of the Internet Information Server - is responsible for looking up reflection information within assemblies and generating a SOAP/WSDL interface to the method add over HTTP.

The essence of annotations is that information is stored together with the code so that some other meta-program will need only the executable file to access the information. Although this may seem to be a little change with respect to configuration files shipped with the executable program, it makes all the difference. With annotations the programmer can decorate the program, without having to define bindings between types and custom information. Moreover configuration files are separated from the executable, leading to a weaker link between the code and its configuration. In the past we have dearly paid the separation of the meta-data from the data, as it is still witnessed by the COM [Rog97] architecture in Windows, where meta-data are stored inside the disliked system registry.

To better appreciate the effectiveness of custom annotations versus the use of external configuration files it is worth to briefly describe the Java Web Service development pack [Javb], currently based on Java 1.4 (the Java version prior to custom annotations). With this library the programmer should define several XML configuration files to control the module responsible for generating SOAP/WSDL. For instance the interface of the Web service is defined with an XML document similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service
      name="MyHelloService"
      targetNamespace="urn:Foo"
      typeNamespace="urn:Foo"
      packageName="HWS">
      <interface name="HWS.HelloWSIF"/>
  </service>
</configuration>
```

Despite its verbosity, to annotate the `HWS.HelloWSIF` interface as a web service (i.e., all the methods of the interface should be considered operations of the service) is the only purpose of the file.

3 Using annotations

Libraries were originally conceived as collections of common-use routines that programmers can import within their programs. Today libraries have become tangled set of programming abstractions (usually in the form of classes) modeling some application domain. To use a library it is required to understand its lingo and how the domain values and operations fit together.

Often libraries are used as a way to extend the programming language with new features (this practice originated with C where even the basic I/O was provided in the form of a library); in a sense they contribute to define a language within the language, designed for a given application domain.

In this section we discuss possible uses of custom annotations to support the definition of library interfaces.

3.1 General considerations

Custom attributes allow tagging programming elements; they differ from inheritance in two ways:

1. annotations are parametric, inheritance no (unless some form of generics is taken into account, and even then it is possible only if specialization is available);
2. unlike inheritance that imposes a small amount, though not null, of overhead at run-time, annotations are passive unless explicitly read

Another important aspect of annotations is that they are orthogonal to other relations; therefore they are suitable for introducing new relations among types of a programming language. Attributes are user-defined, thus there is not a predefined set of them, and a library may introduce as many of them as required.

In the area of domain specific languages custom attributes are useful to define the traits of types [CE00]. Traits are used to configure a generic library so that the amount of information is enough to specialize it to some particular application. In the context of generative programming traits are usually processed at compile time, along with program specialization. At the moment custom annotations are processed at run-time, introducing possible overheads that could be in principle avoided. We will discuss further this issue in the next section.

Custom annotation cannot refer directly objects that will be available at run-time. This is required because they should be processed at compile time, in a different context of the compiler.

3.2 Serialization

Serialization is the process of writing a structured object in a serial stream. As we pointed out in the introduction serialization originated the idea of using interfaces for tagging classes in Java.

With custom attributes it is possible to go further and control the whole process of serialization of instances of a given class. Let us consider the following example:

```
[XmlRoot("NewGroupName"), XmlType("NewTypeName")]
public class Group{
  [XmlArrayItem("MemberName")]
  public Employee[] Employees;
}
```

In this case the class Group has been annotated to indicate how its instances should be serialized. The root element will be named as indicated, the same will happen for XML type name that will be used within the associated XSD schema. More interesting is the annotation over the Employees field, which indicates that in the serialized array only the `MemberName` fields of `Employee` instances must be serialized. Thus in the serialized structure we will only partially serialize the associated employees.

3.3 Indigo and Web Services

We already discussed in the previous section how attributes can be used for defining Web services. A class defines a Web service, and annotated methods indicate the methods that should be exposed as operations.

The upcoming library codenamed Indigo [Win] (now dubbed as Windows Communication Framework) for supporting distributed computations based on web services standards heavily rely upon custom annotations. The library revolves around the notion of *data contract* and *service contract*. As we might guess from the names, the first refer to the structure of the data as it is seen from outside of the application, the second to the definition of published operations.

Here is a simple example of data contract:

```
[DataContract]
public class Person {
  [DataMember]
  public string fullName;
  [DataMember]
  private int age;
  private string mailingAddress;
  private string telephoneNumberValue;
  [DataMember]
  public string TelephoneNumber {
    get { return telephoneNumberValue; }
    set { telephoneNumberValue = value; }
  }
}
```

The traditional approach to marshalling in frameworks like CORBA [COR], Java RMI [Gro01], and .NET remoting [MNW02], is to define a type so that its serialized form coincides with the message to be sent on the network in inter-process communications; in this way we let the run-time take care for us of the communication.

Using custom attributes Indigo decouples the data structure from its serialized form required for network communications. This is possible because, as we already said, custom attributes defines an orthogonal dimension to that of the type system.

In the example above only the members labeled `DataMember` will be serialized in communications (even if they are private inside the process!). The same approach is used for defining data contracts:

```
[ServiceContract]
public interface IOne {
  [OperationContract(IsOneWay=true)]
  void A();
}
```

Service contract provide information about how methods should be exposed to network users of the service. Annotations allow us to provide additional information on the behavior of the particular operation, in this case the fact that the operation will not return any value so that the client can close the connection as soon as possible. A similar approach has been taken by Robotics4.NET [CCEP05], a software library supporting the development of control software for robotics systems. In this case annotations are used to define incoming and outcoming messages from a sort of agent, called roblet. Custom annotations are used by the framework to implement the communication infrastructure among the roblets and the control software of the system. The following is an example of such roblet:

```
namespace HeartBeat {
  public class Beat : RobletMessage {
    public long tick = DateTime.Now.Ticks;
  }

  [OutputMessage(typeof(Beat))]
  public class HeartBeatRoblet : Roblet {
    public HeartBeatRoblet() : base("HB") {}
    protected override void Run() {
      SendState(new Beat());
    }
  }
}
```

The `SendState` method is responsible for taking care of message dispatching, and it its behavior is controlled by the custom annotations indicating friendship among agents, input and output message types.

3.4 Relational Interface to Databases

In [AC02] it is discussed how to extend C++ with reflection support by means of template meta-programming techniques. The proposed reflection system provides support for custom meta-data.

In the paper it is discussed how a library for building search engines can benefit from the declarative power of custom attributes. In this case attributes drive storage information of the objects:

```
class DocInfo {
  char const* name;
  char const* title;
  int date;

  META(DocInfo,
    (FIELD(name, (MaxLength(256),
                  IndexType(Index::primary))),
     FIELD(title, MaxLength(2048)),
     FIELD(date, IndexType(Index::key)))
  );
};
```

In a way similar to C# attributes are objects stored within the meta-class. In this example we use `MaxLength` and `IndexType` attributes to control how the search engine library must store and index objects on the secondary storage.

3.5 Code Annotations

Assuming custom annotations capable of annotating portions of code as it is done in [a]C# [CCC05], an extension to the C# language, we can use them for more finer grain tasks.

Using this kind of annotations it is possible to annotate a code with hints on about how to produce the concurrent version of it:

```
public void m() {
  [Parallel("Begin of a parallel block")] {
    Console.WriteLine("Main thread code");
    [Process("First process")]{ /* Computation here */ }
    [Process]{ /* Computation here */ }
  }
  Console.WriteLine("Here is sequential");
}
```

In this case we rely on annotations to mark `Parallel` a block of code. Inside we define code blocks annotates as `Process` that can run in parallel.

3.6 Attribute Usage

Microsoft .NET defines a set of “meta-attributes” that can be used as annotation when defining an attribute class. These annotations are used to possibly constraint the attribute usage. The following example defines an attribute that can be used only once and only on classes:

```
[AttributeUsage(AttributeTargets.Class,
                AllowMultiple=false)]
class ClassTgtAttribute : Attribute {}
```

In a sense, the ability of specifying that an attribute can be used only on classes or methods, if it is inherited or not, provides a means for specifying a sort of a customizable syntax for custom attributes.

3.7 Designer Environments

Microsoft Visual Studio [Mic] designer is capable of loading arbitrary components during the design process of user interfaces. At design time components are configured by specifying a subset of properties that the component should have at run-time.

Microsoft .NET controls can indicate to the designer which properties can be configured at design time by means of custom attributes. Default values of design-time properties are also specified through custom attributes.

The designer is able to display a preview of the component while designing an interface. A custom attribute specifies which class is responsible for generating the preview of a component. The designer, however, should inherit from a specific class in order to be eligible for its role.

Java designer also relies on reflection information in order to load components into the designer. However, in this case a naming convention is used to determine properties so be shown inside the designer. The naming conventions used by Java are defined by the Java Beans specifications.

3.8 Final Considerations

In this section we presented several applications of custom attributes. We believe that many others are possible, making extensible meta-data an important tool in the library-designer toolbox.

In particular we believe that the declarative aspect of the approach allow library developers defining interfaces both operational and declarative.

Custom attributes have almost no drawbacks: they allow defining arbitrary relations among data types, are distributed with executables, and always accessible through the reflection API. However there is a noticeable exception: there is the risk of a possible overhead, due to the facet that meta-data interpretation is often performed at run-time. In the next section we will discuss this aspect of the problem.

4 About performance

Performance is always important, and custom attributes should not impose a significant overhead over a computation in order to be really used.

At a first glance it might be evident that meta-data can be retrieved only at run-time through reflection. This implies that, if attributes are used to specify traits of a library, we must postpone computations that could be done at compile time, at run-time.

This is true for the examples shown in the previous section. However it is not true in general: meta-programs can be run before the so-call “run-time”, though they run after the compiler. It is the case of several tools that manipulates binaries available for the various virtual machines.

Nevertheless, when we are interested in using custom attributes directly at run-time, we must consider that the time spent for reading meta-data is not zero. It is however possible to drown this overhead into the overall computations costs: for instance, the Microsoft XML serializer, for instance, dynamically generates a class for each type it serializes, and annotations are read during this generation process. After this generation phase serialization takes place without any more accesses to custom meta-data.

5 Conclusions

In this paper we have discussed how custom annotations may affect the design of libraries. The main impact of the mechanism is at the level of library interface; however it also influences the internal design of the library.

Custom annotations provide a mean for library users to declare their intentions, and for library developers to better adapt to different uses of the library. If used in their simplest form annotations require to be processed at run-time. The overhead imposed for accessing them is in general not significant, though it is possible to get rid of it by executing a meta-program responsible for processing annotations before that the program is executed.

We believe that custom annotations will play a significant role in the design of libraries in the next years, and they will be added to other programming systems that still lacks of this kind of support.

6 References

[AC02] G. Attardi and A. Cisternino, *Self reflection for adaptive programming*, Proceedings of Generative Programming and Component Engineering Conference (GPCE), LNCS 2487 (2002), 50–65.

[CCC05] Walter Cazzola, Antonio Cisternino, and Diego Colombo, *Freely Annotating C#*, Journal of Object Technology **4** (2005), no. 10, 31–48.

[CCEP05] A. Cisternino, D. Colombo, G. Ennas, and D. Picciaia, *Robotics4.NET: Software body for controlling robots*, IEE Proceedings Software **152:5** (2005), 215–222.

[CE00] K. Czarneki and U.W. Eisenacker, *Generative programming - methods, tools and applications*, Addison-Wesley, 2000.

[COR] *Corba web site*, available at http://www.corba.org/, Accessed: 20/3/2006.

[ECM] *Ecma 335, common language infrastructure (cli)*, available at http://www.ecma.ch/ecma1/STAND/ecma-335.htm, Accessed: 20/3/2006.

[Gro01] W. Grosso, *Java rmi*, O'Reilly, 2001.

[Java] *Java web site*, available at http://java.sun.com/, Accessed: 20/3/2006.

[Javb] *Java web service development pack web site*, available at http://java.sun.com/webservices/reference/index.html, Accessed: 20/3/2006.

[Mic] *Microsoft visual studio web site*, available at http://msdn.microsoft.com/vstudio/, Accessed: 20/3/2006.

[Mil03] J. Miller, *Common language infrastructure annotated standard*, Addison-Wesley, 2003.

[MNW02] S. McLean, J. Naftel, and K. Williams, *Microsoft .net remoting*, Microsoft Press, 2002.

[Rog97] D. Rogerson, *Inside com*, Microsoft Press, 1997.

[Win] *Windows sdk web site*, available at http://windowssdk.msdn.microsoft.com/library/, Accessed: 20/3/2006.

Framework design using inner classes - Can languages cope?

Kasper Østerbye and Thomas Quistgaard
IT University of Copenhagen
Rued Langgaardsvej 7, 2300 Copenhagen, Denmark
kasper@itu.dk, tquistgaard@itu.dk

ABSTRACT

Inner classes have been part of the Java specification since version 1.1, and are an integral part of the Beta language. In Java they have *primarily* been used in connection with event handling in the user interface libraries. This paper investigates inner classes as the cornerstone in the architecture for the layout part of a GUI framework, and how the two languages support this architecture. The difference in support in the two languages is shown to have clear impact on the usability of the framework for application programmers. While Java has come a long way, it turns out that three obstacles need to be removed to fully support the architecture. Of these, it should be straight forward to address two of them in Java. The third lies in the realm of aspect oriented programming. The proposed architecture is itself interesting, as it provides an insight into larger-scale use of inner classes, and provides a compiler supported idiom for the implementation of the composite design pattern.

1. BACKGROUND

In object oriented programming languages which support inner classes, e.g. Simula [Dahl *et al.*, 1968], Beta [Madsen *et al.*, 1993] and Java [Gosling *et al.*, 2005], we have come across an interesting implementation idiom which relates to the composite pattern [Gamma *et. al.* 1995]. The design is that the lexical nesting mirrors the composition of the objects. A simple example in Java is:

```
class Menu{
    private String name;
    private List<Item> items = new ArrayList<Item>();
    public Menu(String name){
        this.name = name;
    }

    protected abstract class Item{
        private String name;
        public Item(String name){
            this.name = name;
            items.add(this);
        }
        abstract void action();
    }
}
```

The important part is that the class Menu has a list of Items, and each Item adds itself to the menu when created. The usage of generic collections does not play any role in the discussion. The class Item is declared protected, so it can only be used in subclasses of Menu.

The declaration of enables the following client code:

```
Menu editMenu = new Menu("Edit"){
    Item copy = new Item("Copy"){ void action(){…} };
    Item cut = new Item("Cut") { void action(){…} };
    Item paste = new Item("Paste") { void action(){…} };
}
Menu fileMenu = new Menu("File"){
    Item quit = new Item("Quit") { void action(){…} };
    …
}
```

The implementation uses anonymous inner classes as a concise implementation of the singleton. In addition, lexical scoping avoids parsing a menu as parameter to items when they are created.

The idiom enables a somewhat declarative style, where the physical structure of the menu is mirrored in the program layout itself. Other examples of this structure are the relationship between rule-set and individual rules, where the rules can be defined inside their rule-set; or the hierarchical structuring of a GUI, to which we will return. In Beta, we have also applied the idiom in the area of process composition [Østerbye & Kreutzer, 1999].

In general, there are a number of qualities which one would like a framework to have:

- It should be simple to use for the application programmer
- Misuse of its constructs should be captured at compile time
- Its application should be concise
- The framework should be extensible
- The underlying implementation should perform adequately – that is, should not constitute a bottleneck in the application

Also, it is important to realize that a framework is a generalization over a *set* of applications. There are therefore (interesting) applications that are covered and other that are not covered by the framework.

Compared to the above qualities, the simple menu illustrates some important points:

- Simplicity. The application programmer need not have an explicit set of statements which associates items to menus. The menu structure is manifest in the program structure.
- Compile-time checks. Attempting to use Items outside the scope of a Menu will not work, class Item has been declared protected, and can therefore only be seen in subclasses of Menu. The compiler checks this (but will give un-informing error-messages in case of violations).

- Conciseness. There is not much extra information except the definition of the hierarchical structure between Menu and Item. There is some redundancy (Item and item name repeated twice), which we will return to.
- Lack of flexibility. These qualities have been obtained at the cost of not being able to dynamically change which menu a given item belongs to.

In the remainder of this paper this idiom will be further elaborated. The next section introduces the framework we have developed to investigate the idea. The description highlights the inner class idiom, and the problems we have encountered in implementing the idiom in Java. Then we contrast some of the problems encountered in the Java solutions with similar (but less problematic) solutions in Beta. We end with a summary of our findings.

2. HIERARCHICAL GUI LIBRARY (HGL)

Before examining the differences between the languages, a slightly more complex example is needed. As part of his master's thesis, Thomas Quistgaard [Quistgaard, 2005] designed and implemented a hierarchical user interface framework. The goal was to apply the above inner class idiom for a full scale framework to achieve a simpler to use GUI framework than say Swing, which is notorious for its complexity.

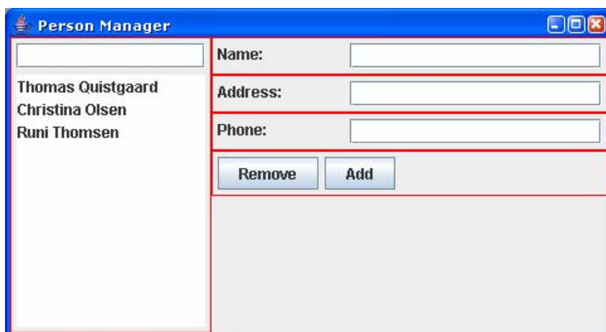
The design is based on a few overall guiding principles:

a) The program structure should mirror the hierarchical structure of the GUI.

b) The components are added to their enclosing container in the order they are declared (Using the same idiom as with the menu).

c) The physical layout is declared using annotation types, to provide a clear separation between hierarchical structure and physical layout, and to provide a path for later tool manipulation of physical layout.

To explain the design, a simple example will be used.



The above Frame (top level window) has to its left a text field in which one can enter a search string. All persons that contain the string in their name are shown in the list below. Selecting a person brings up the underlying data in the right part for examination or modification. The above GUI is defined in HGL as shown below.

This example illustrates the three design principles. Every graphical object which appears inside the frame is declared as member fields of the anonymous personManager Frame. Anonymous inner classes give a syntactic structure which enables the hierarchical structure to follow the program structure. Inside the frame, a panel is declared, inside which a text field and a list are declared.

The components are added in the order they are declared. Inside listpanel, a TextField, and then a List are declared. These are added to the panel in the order of declaration.

```
Frame personManager = new Frame("Person Manager") {

    @Vertical
    Panel listpanel = new Panel() {
        @Width(150)
        TextField searchtextfield = new TextField();
        @Width(150) @Height(300)
        List list = new List(); // GUI list, not a collection library List
    };

    @Vertical @Padding(0)
    Panel infopanel = new Panel() {

        @Horizontal
        Panel namepanel = new Panel() {
            @Width(100)
            Label namelabel = new Label("Name:");
            @Width(200)
            TextField nametextfield = new TextField();
        };


        @Horizontal
        Panel addresspanel = new Panel() {
            @Width(100)
            Label addresslabel = new Label("Address:");
            @Width(200)
            TextField addresstextfield = new TextField();
        };

        @Horizontal @Hlock(false)
        Panel phonepanel = new Panel() {
            @Width(100)
            Label phonelabel = new Label("Phone:");
            @Width(200)
            TextField phonetextfield = new TextField();
        };

        @Horizontal @Hlock(false)
        Panel addpanel = new Panel() {
            Button removebutton = new Button("Remove");
            Button addbutton = new Button("Add");
        };
    };// end infoPanel
}; // end personManager
```

Frames and Panels are containers that contain other components, including other Panels. Layout is defined using annotations. Annotations are user defined metadata. Syntactically, annotations are located as modifiers, in front of the element they annotate. Annotations are accessed programmatically through reflection. @Horizontal is a user defined annotation, which is used to specify that the layout in the panel should be horizontal instead of the default vertical. Annotations can include simple values as parameter. @Padding indicates the space between a component and the previous component in the same container (or the border if it is the first). Hlock and Vlock indicate resizing behaviour.

2.1. Addressing components

The above code does not specify behaviour, only layout. There are two kinds of behaviour which is interesting in connection with GUI frameworks: tying the GUI to the application data and business logic, and

(our focus) ensuring graphical consistency. If we select "Christina Olsen" and modify her name, that name change ought to be reflected not only in the application data, but also in the list. Standard Swing list listeners raise an event if an element is being added or deleted from a list, but not if an element is changed. A direct approach would be to let the textChanged event from the nametextfield directly change the list:

```
TextField nametextfield = new TextField(){
    void onChange(TextChangeEvent e){
        listpanel.list.changeSelected(this.getText());
    }
}
```

But this does not work because listpanel is of type Panel, and Panel does not have a field named list, though the concrete object listpanel refers to does indeed have this field. To get around this, we implemented a method get (using reflection), which allow us to write the above code as:

```
TextField nametextfield = new TextField(){
    void onChange(TextChangeEvent e){
        ((List)get("listpanel.list")).changeSelected(this.getText());
    }
}
```

Unfortunately, we can no longer check at compile-time that the path exist and is spelled correctly.

2.2. Compile-time checking

The hierarchical definition of the components plays an important role in making certain that the compiler can catch as many mistakes as possible.

At the outset, the design looks like a composite pattern, with the components as leafs, and panel and frame as composites. In Swing, a Frame is a top-level window, and as such:

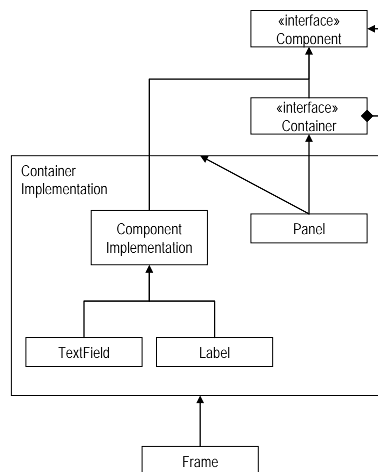
- No component exists outside a frame
- No frame can be put inside a frame (a frame is not a component)

The standard composite pattern does not treat the issue of a dedicated root composite. In particular, no compile time checks are carried out.

To provide a compile time checkable version of the rooted composite pattern, we can again use inner classes as a key:

```
interface Component{...}
interface Container extends Component{
    List<Component> getComponents();
    void addComponent(Component c);
}
class ContainerImplementation {
    private List<Component> components;
    public Component getComponents(){ return components; }
    public void addComponent(Component c){ components.add(c);}
    protected class TextField implements Component{...}
    protected class Label implements Component{... }
    protected class Panel extends ContainerImplementation
                                        implements Container{... }
}
public class Frame extends ContainerImplementation{
    ...
}
```

The interface for components has a specialized interface which represents containers. The container interface specifies that it consist of components, whereby we obtain the usual recursive composite pattern.



The class ContainerImplementation contains the necessary infrastructure to implement the Container interface, but does not declare that it does so (no implements clause). ContainerImplementation has two subclasses – Panel and Frame. Frame is a public class, and can be used as expected. However, it does not implement the Container interface; hence Frames are not components and cannot be contained in a container. Panel on the other hand declares that it implement the Container interface. The methods to do so are inherited from ContainerImplementation.

Leaves (e.g. TextField and Label) and Panel are protected inner classes of ContainerImplementation, and can therefore only be used in subclasses of ContainerImplementation, whose only public subclass is Frame. Within a concrete Frame, e.g. the anonymous class assigned to personManager, one has access to the protected inner classes TextField, Label and Panel.

The design does not change the way how the application programmer uses the framework. And it achieves the two compile-time checks we wanted:

- It is not possible to add a frame inside a frame, as a Frame is not a component.
- It is not possible to use any components outside a frame, since they are protected inner classes of ContainerImplementation (allowing components to be used both within Frame and Panel).

We find this idiom for implementation of the composite design pattern a contribution in its own right. It gives a solution to the notion of a special root composite, and it can enforce this design at compile-time.

3. IS BETA BETTER?

The original idea for the HGL framework originates in Beta [Lidskjalv, 2002]. In [Quistgaard, 2005] the design is expanded, in particular by adding declarative layout, and accommodating the design to fit Java. In this section we will examine a few issues where Beta and Java differ and how this impact the details of the library.

3.1. Variable declaration syntax

A mundane difference between Java and Beta relates to how variables are defined. Java declares variables as "Type varName", whereas Beta does "varName: Type".

In Beta, the declaration of editMenu would look like (Though we use {…} instead of Beta's (#...#)):

```
editMenu:@Menu{
      copy :@Item{ action{...} };
      cut:@Item{ action{...} };
      paste:@Item{ action{...} };
}
```

The Beta syntax is more concise, as we avoid stating Item twice, both as type name and after new. Furthermore, we do not need to pass the name of the menu item as parameter, as we are able to pick out the name through reflection.

A more important consequence of the different way of declaring the variables surfaces in connection with the addressing of components. In Java, it was necessary to use reflection in addressing components, although we were able to hide this in the get method. In Beta, the type of the corresponding listPanel variable is the actual type which does have a list field. Hence, in Beta we are able to avoid reflection in connection with field addressing.

3.2. Reflection

Reflection is typically avoided because of bad performance and because it postpones checks to runtime. In the concrete design of HGL, event handling was not done as described earlier. Instead a mapping between events and handler methods were established programmatically as:

```
((List)get("listPanel.list")).onSelect("listElementSelected");
```

A series of such statements are executed at initialization. The get method returns a list, and this list is told what method to execute when an element is selected.

Because of reflection, the spelling error "listPanel.list" (should be "listpanel.list") is first caught at runtime. However, the mistake is caught at program initialization. Thus, running the program just once will reveal the error. Efficient run-time structures are constructed during initialization, so no execution time is lost in practical usage.

3.3. Annotation checks at compile time

It would have been compelling to use the new annotation processing tool [APT 5.0] to check that annotations were used correctly, for instance that the Vertical annotation is only associated with Panels. However, neither Java reflection nor APT allows us to access inner classes; hence we cannot traverse structures of anonymous inner classes. So while we basically have all the machinery in place, a design choice in Java and APT prevents us from doing compile-time checks in connection with applications of our framework.

3.4. Object initialization

Java has two ways to provide information when an object is created. One can pass parameters to its constructor, and sometimes one can attach annotations to its declaration. In Beta one cannot do either one.

So, the best approximation one can do for the layout information in Beta is something like the following

```
listPanel:@Panel{ Layout::Horizontal; Padding::{do 0->padding};
      ...
}
```

This syntax specifies that listPanel is a constant which refers to an object which is a subtype of Panel, where the virtual type Location is bound to Horizontal. This corresponds roughly to giving Horizontal as a type parameter. The method Padding is specialized to return the value 0.

There are several drawbacks compared to the annotation approach

- Annotations are well suited for tool manipulation.
- The specification of concrete values, like 0 padding becomes quite clumsy.

But there are a number of drawbacks associated with the annotation approach we have used as well.

1) One cannot associate annotations with anonymous inner classes. Hence we have been forced to annotate the fields instead.
2) Annotations need to be manipulated through reflection, which implies poor performance. In our case, however, it is only done when the Frame is initialized, not when the GUI is used.

3.5. Framework extension

The complex design makes it hard to add new component types to the library, as we effectively need to add new definitions inside a package protected class. The Beta compiler supports a Fragment system. The fragment system is a way to declare insertion points in classes, and enables libraries of code which, at compile-time, is weaved into these insertion points. A problem in HGL is that one cannot add new protected component types to the ContainerImplementation class. A Java version of Beta's fragment system would allow us to write the ContainerImplementation class as:

```
class ContainerImplementation {
      private List<Component> components;
      ...
      protected class Panel extends ContainerImplementation implements Container{
            ... }
      «SLOT ExtraComponents: Declarations»
}
```

A component, e.g. GanttChart, can be written, specifying that is intended to be inserted at the ExtraComponents slot. GanttChart is compiled as if it were lexically located at that slot, with access to all the same lexical information as the standard components.

To use GanttChart, in your application, you declare it in an insert clause. Rather than making GantChard available in the global name space, insert makes GandChart available as if it were inserted into the slot. Hence, GanttChart cannot be used outside of Frames. On the other hand, it is readily available to be used as any other components.

It is highly unlikely that such a mechanism should be included in Java. A similar effect can be obtained using aspect oriented programming. The idea is to use insertion to place the GanttChart into the ContainerImplementation as:

```
aspect MyComponentLibrary {
      protected class ContainerImplementation.GanttChart {...}
}
```

At present, however, the most widely used aspect compiler for Java, AspectJ [AspectJ, 5.0] does not support insertion of inner classes, and aspect oriented programming tends to focus on other issues than insertion. The difference between the slot approach and aspects is discussed in [Ernst, 2000]. For our needs there is no fundamental difference.

While C# does not support inner classes, its notion of partial classes is also a solution. If ContainerImplementation were partial, it could extended it with new components. Partial inner classes have to be worked out in practice. The slots in Beta can only be used for adding new classes and

methods, not new fields, as that would change the size of objects, which would prevent separate compilation.

4. SUMMARY

With some tradeoffs, we have been able to implement HGL in Java. Its design, however, is cleaner in Beta. In particular we have encountered three major problems in Java:
First, the problem with the type of variables and anonymous inner classes in Java is a hindrance for our design of HGL. One solution is to adopt the val type from ML, to state that the type of a variable should be deduced by type inference. Hence, out listpanel should be defined as:

```
final val listpanel = new Panel(){
    ...
    List list = new List();
}
```

This way the type of listpanel could be the anonymous subclass of Panel which has the field list, so it can be compile-time checked that listpanel.list is indeed a legal object path.

While it is unknown if such a val construct will make it to Java, a variation which can solve the problem will be available in next version of Visual Basic.

Second, the standard java annotation processing tool allows us to write our own compile-time modules. This facility is intended for writing code-generators in connection with J2EE. However, it is tempting to view it as general compiler extension mechanisms, which allow us to write custom compile-time checks for the usage of libraries and frameworks. In its present state, however, we cannot use it for HGL. Nevertheless, a possible example might be the unit testing framework JUnit [JUnit]. JUnit assumes certain naming conventions, which are checkable using reflection, and can also be checked at compile time. We have not investigated this further. But in our case, neither reflection nor APT allows us to examine the whole program; in particular anonymous inner classes can not be traversed.

Thirdly, to make the inner class approach presented here feasible, it is necessary to solve the problem of adding inner classes to an existing class. Java needs to be extended with something similar to partial classes, or AspectJ needs to be able to handle introductions of inner classes. The notion of MixIn Layers [Smaragdakis & Batory, 1998] provides another view on how the existing framework can be refined into a new framework with additional components. Their solution provides the necessary infrastructure we ask for, but from our experience with HGL we do not necessarily need all the capabilities of MixIn Layers.

Scala [Odersky *et al.*, 2005] provides the key mechanisms needed to implement the inner class idiom as well, in particular object definitions and anonymous inner classes. However, it seems that Scala has the same problem as Java when it comes to extending the framework, and it is not clear what mechanisms can be used to separate logical and physical layout.

Compared to [Hedin & Knudsen, 1999] we are applying some of the mechanisms from Beta that they describe as providing benefit for framework design. In relation to their work, the contribution in this paper has been to apply those guidelines in the context of a Java based framework, and to report where Java fails in achieving the goals. However, an important issue for framework design not mentioned in [Hedin & Knudsen, 1999] is object initialization. Here Java is superior to Beta, providing both field initializers and annotations.

Of the major object oriented languages, it is only Java that supports inner classes. C# and C++ share a design, in which a class can be defined inside an other class, but the inner class will not have instances of the outer class as lexical scope for its objects, hence not even the simple Menu-Item example will work. Eiffel, Smalltalk and many other languages do not even allow the simple nesting of C++ and C#.

5. REFERENCES

[APT 5.0] Annotation Processing Tool (apt), part of Java 2 Standard Edition. http://java.sun.com/j2se/1.5.0/docs/guide/apt/index.html

[AspectJ, 5.0] AspectJ Project. http://eclipse.org/aspectj/. Accessed October 3rd, 2005.

[Dahl *et al.*, 1968] O.J. Dahl, B. Myrhaug, K. Nygaard. *SIMULA 67 Common Base Language*. Norwegian Computing Center, Oslo, 1968.

[Ernst, 2000] Erik Ernst Syntax based modularization: invasive or not? in Tarr, P., Bergmans, L., Griss, M. and Ossher, H. (eds.), *Workshop on Advanced Separation of Concerns* (OOPSLA'00). Department of Computer Science, University of Twente, The Netherlands.

[Gamma *et. al.* 1995] Eirich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[Gosling *et al.*, 2005] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification (Third Edition).* Addison Wesley 2005.

[Hedin & Knudsen, 1999] Görel Hedin and Jørgen Lindskov Knudsen. Language Support for Application Framework Design. In *Implementing Application Frameworks: Object Oriented Frameworks at Work.* Ed. M.E. Fayad, D.C. Schmith, R.E. Johnson. Wiley 1999.

[JUnit] Unit testing framework for Java. http://www.junit.org/index.htm

[Lidskjalv, 2002] *Lidskjalv: User Interface Framework – Tutorial.* Mjølner Informatics Report, MIA 95-30, February 2002. http://www.daimi.au.dk/~beta/mjolner_system/lidskjalv.html

[Madsen *et al.*, 1993] Ole Lehrman Madsen, Birger Møller-Pedersen, Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison Wesley 1993.

[Odersky *et al.*, 2005] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *The Scala Language Specification Version 1.0*. Accessed from http://scala.epfl.ch/docu/index.html October 3, 2005.

[Quistgaard, 2005] Thomas Quistgaard. *The Hierarchical Graphical Library.* Masters Thesis, IT University of Copenhagen, 2005. http://hgl.sourceforge.net. (in Danish)

[Smaragdakis & Batory, 1998] Yannis Smaragdakis and Don Batory. *Implementing Layered Designs with Mixin Layers*. Proceedings of ECOOP'98, Brussels, Belgium, July 1998. Lecture Notes in Computer Science 1445, Springer-Verlag.

[Østerbye & Kreutzer, 1999] Kasper Østerbye and Wolfgang Kreutzer. *Synchronization abstraction in the BETA programming language.* Computer Languages 25 (1999) 165-187.

The Diary of a Datum: An Approach to Modeling Runtime Complexity in Framework-Based Applications

Nick Mitchell
IBM TJ Watson Research Center
19 Skyline Drive
Hawthorne, NY USA
+1 914-784-7715
nickm@us.ibm.com

Gary Sevitsky
IBM TJ Watson Research Center
19 Skyline Drive
Hawthorne, NY USA
+1 914-784-7619
sevitsky@us.ibm.com

Harini Srinivasan
IBM Software Group
Route 100
Somers, NY USA
+1 914-766-1885
harini@us.ibm.com

ABSTRACT

In large-scale framework-based applications, every piece of information has a complex story to tell about its journey. As it makes its way through a tangle of reusable frameworks, it may be transformed from a string, to an Integer, to an integer, and finally to a date. Over the past several years, our research group has analyzed dozens of industrial, framework-based applications. Often, simple functionality requires a seemingly excessive amount of runtime activity and complexity. We have found it increasingly difficult to understand behavior, weigh design tradeoffs, and assess if and how performance problems can be fixed.

Much of this activity revolves around the transformation of information from one form to another. In this paper we present an approach to understanding runtime behavior that models activity as the flow of logical content through a sequence of transformations. We show how to manually group and filter activity into a hierarchy of data flow diagrams, to make an otherwise overwhelming amount of information about a run manageable. We give a detailed example that illustrates the approach, and also demonstrates the complexities typically found in this class of application. We show how structuring behavior according to transformations allows us to introduce new metrics of cost and complexity derived from the topology of the diagrams.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *complexity measures*

General Terms

Measurement, Performance, Design

Keywords

Dynamic analysis, program understanding, complexity assessment, performance analysis, design recovery

1. INTRODUCTION

Large-scale applications are being built from increasingly many reusable frameworks, such as web application servers (that use SOAP [5], EJB, JSP), portal servers, client platforms (Eclipse), and industry-specific frameworks. Over the past several years, our research group has analyzed the performance of dozens of industrial framework-based applications. In every application we looked at, an enormous amount of activity was executed to accomplish simple tasks. This was the case, even after some tuning effort has been applied. For example, a stock brokerage benchmark [10] executes 268 method calls and creates 70 new objects just to move a single date field from SOAP to Java. Beyond identifying bottlenecks, this paper presents an approach to making visible the nature of runtime complexity and inefficiency in these applications.

In our experience, inefficiencies are not typically manifested in a few hot methods. They are mostly due to a constellation of transformations. Each transformation takes data produced in one framework and makes it suitable for another. Problems are less likely to be caused by poor algorithm choices, than by the combined design and implementation choices made in disparate frameworks. In a web-based server application, for example, the data arrives in one format, is transformed into a Java business object, and is sent to a browser or another system – e.g. from SOAP, to an EJB, and finally to XML. Surprisingly, inside each transformation are often many smaller transformations; inside these are often yet more transformations, each the result of lower-level framework coupling. In addition, many steps are often required to *facilitate* these transformations. For example, a chain of lookups may be needed to find the proper SOAP deserializer. In our benchmark example, moving that date from SOAP to Java took a total of 58 transformations.

How do we know if 58 transformations is excessive for this operation? And if so, what could possibly require so many? Traditional performance tools model runtime behavior in terms of implementation artifacts, such as methods, packages, and call paths [1,2,3,7,8,18]. Transformations, however, are implemented as sequences of method calls, spanning multiple frameworks. In this paper, we present an approach for understanding and quantifying behavior in terms of transformations. We believe this model enables:

- Evaluation of an implementation to understand the nature of its complexity and costs, and assess whether they are excessive for what was accomplished.
- Comparison of implementations that accomplish similar functionality, but use different frameworks or physical data models.

We model the behavior of a run by structuring it as the flow of data through transformations. We believe that structuring in terms that are abstracted from the specifics of any framework will enable new ways of evaluation and comparison. We briefly show how new cost and complexity measures can be derived from this

model. Generating a model and computing metrics are currently manual processes; parts are amenable to automation in the future. We now describe the approach in more detail.

Structuring Behavior: There often are multiple *physical representations* of the same *logical content.* For example, the same date may be represented as bytes within a SOAP message, and later as a Java Date object. Our approach structures runtime activity as data flow of logical content, as illustrated in Figure 1. We show the data flow as a hierarchy of data flow diagrams [6,9]. Each edge represents the flow of a physical representation of some logical content. Each node represents a *transformation* – a change in logical content or physical representation of its inputs.

Many types of processing can be viewed as transformations. For example, a transformation may be a physical change only, like converting information from bytes to characters or copying it from one location to another; it may be a lookup of associated information, such as finding a quote for a stock holding; or it may be implementing business logic, such as adding a commission to a stock sale record.

It is infeasible to have a dataflow diagram show an entire run. We introduce the concept of an *analysis scenario* that filters the analysis to show just the production of some specified information. We show how to group the activity and data of an analysis scenario into a hierarchy of dataflow diagrams.

Transformation-based Complexity and Cost Measures: We use the number of transformations as an indicator of the magnitude of complexity. We introduce metrics that aggregate based on the topology of the diagrams. For example, 58 transformations to convert one field seems excessive. Knowing that 36 of these occurred at a diagram depth of three indicates that the complexity was due to design decisions were made far from the application code.

We can also aggregate traditional resource costs, such as the number of instructions executed or objects created, by transformation. Aggregating in this new way, as opposed to by method, package, or call path, gives more appropriate metrics of cost for framework-based applications. Throughout the paper we give examples showing the benefits of reporting costs by transformation.

In Section 2 we describe the structuring approach, following the data flow of logical content through transformations. We also discuss strategies for grouping and filtering activity, and give a brief example. In Section 3 we give an in-depth example, that follows our single date field from a SOAP response into a Java business object – a seemingly simple operation with surprisingly complex behavior. In addition to illustrating the approach, this example illustrates the nature and magnitude of the complexities found in large-scale framework-based applications. Structuring by logical data flow also enables new quantitative analyses that can shed light on the costs and complexity of an implementation. In Section 4 we show some of these metrics.

2. STRUCTURING APPROACH

We model runtime behavior using data flow. Using the raw data flow information would give too much, and too low a level of information to make sense of. In this section we present our approach to filtering and grouping activity into a hierarchy of data flow diagrams.

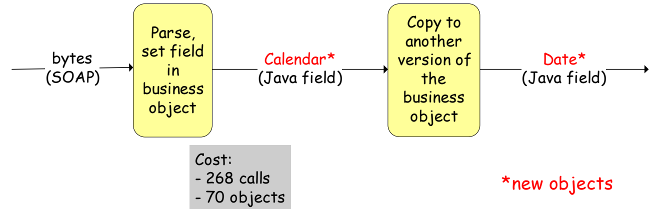


Figure 1: A dataflow diagram of how the Trade benchmark transforms a date, from a SOAP message to a Java object.

Figure 1 shows a dataflow diagram from a configuration of the Trade 6 benchmark [10] that acts as a SOAP client.[1] The figure follows the flow of one small piece of information, a field representing the purchase date of a stock holding, from a web service response into a field of the Java object that will later be used for producing HTML. We follow this field because, of all the fields of a holding, it is the most expensive to process.

Each edge shows the flow of the physical form of some logical content. In the figure, the same purchase date is shown on three edges: first as some subset of the bytes in a SOAP response, then as a Java Calendar (and its subsidiary objects), and finally as a Java Date. Each node denotes a transformation of that data, and it groups together invocations of many methods or method fragments, drawn from multiple frameworks. In Sections 3.3 and 3.4 we discuss in more depth transformations and logical content.

Structuring in this way relates the cost of disparate activity to the data it produced. Figure 1 shows that the cost of the first transformation was 268 method calls and 70 new objects, mostly temporaries.[2] All this, just to produce an intermediate (Java object) form of the purchase date.

2.1 Filtering by Analysis Scenario

The extent of a diagram is defined by an *analysis scenario* that consists of the following elements:

- The output – the logical content whose production we follow
- The physical target of that logical content
- The physical sources of input data
- Optional filtering criteria, such as a specific thread, time interval, or call path

For example, Figure 1 reflects an analysis scenario that follows the production of a purchase date field; its physical target is the Java object that will be used for generating HTML; its physical source is the SOAP message; filtering criteria limit the diagram to just one response to a servlet request, and to the worker thread that processes that request. Note how the filtering criteria allow us to construct a diagram that omits any advance work not specific to a servlet response, such as initializing the application server.

[1] We omit the standard data flow notation for sources and sinks, and instead represent them as unterminated edges.

[2] We used a publicly available application server and JVM. Once in a steady state, we used ArcFlow [1] and Jinsight [7] to gather raw information about the run, *after* JIT optimizations.

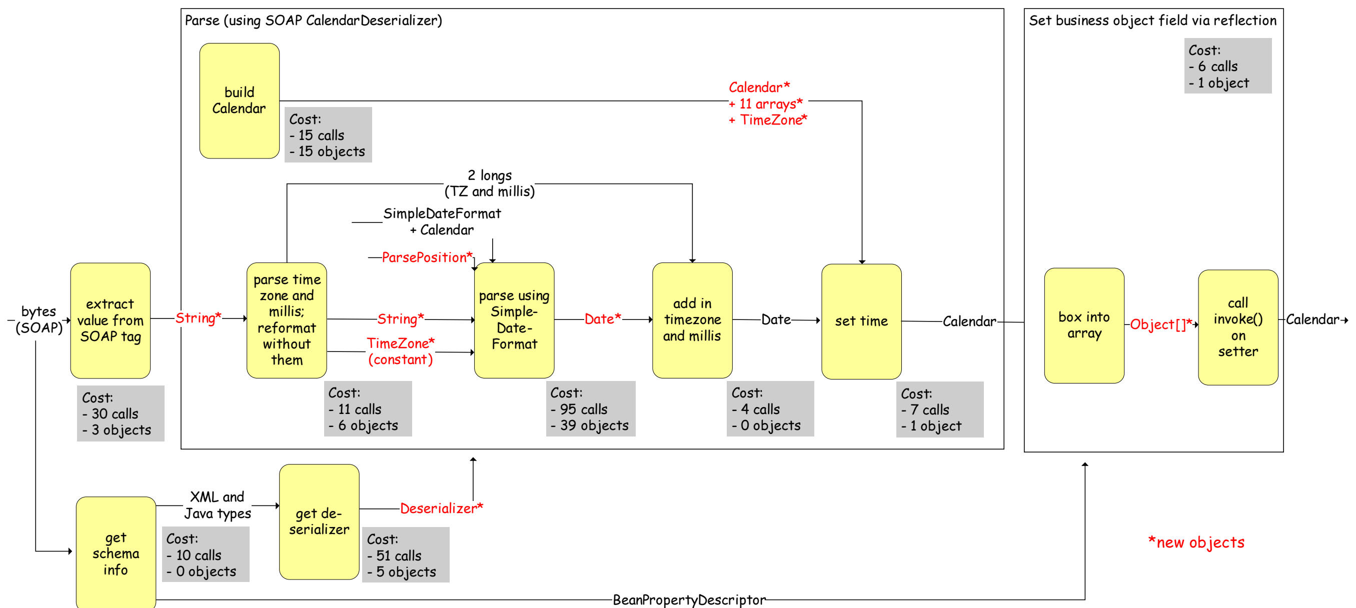


Figure 2. Zooming in on the first step of Figure 1 shows how the SOAP framework transforms the purchase date field.

2.2 Grouping Into Hierarchical Diagrams

Within an analysis scenario, the activity and data could be grouped into data flow diagrams in various ways. In this section we show how we group activity into transformations, to form an initial hierarchy of data flow diagrams. We then apply an additional rule that identifies groups of transformations to split out into additional levels of diagram.

Applications often have logical notions of *granularity* that cut across multiple type systems. For example, a stock holding record, whether represented as substrings of a SOAP message or as a Java object, may still be thought of as a record. Other common examples include fields, subfields, and record sets.

We follow the activity and intermediate data leading to the production of the scenario's output. The top-level diagram shows this at a single level of granularity, that of the output. Each transformation groups together all activity required to change either the logical content or physical representation of its input data. Section 3 gives more precise definitions of logical and physical change. Note that some of the inputs to a transformation will be facilitators, such as schemas or converters. In the diagram for that transformation, we also include the sequence of transformations needed to produce these facilitators. Section 3.1 discusses facilitators in more depth.

While one diagram shows data flow at a single level of granularity, it will also show those transformations that transition between that granularity and the next lower one. For example, the transformation that extracts a field from a record will be included in the diagram of the record.

We form additional levels of diagram to distinguish the parties responsible for a given cost. We define an *architectural unit* to be a set of classes. Given a set of architectural units, a hierarchical dataflow diagram splits the behavior so that the activity at one level of diagram is that caused by at most one architectural unit. The choice of architectural units allows flexibility in assigning responsibility for the existence of transformations. In our experience, architectural units do not necessarily align with package structure. The diagram of Figure 1 shows the field-level activity that the application initiates. Other field-level activity that SOAP is responsible for is grouped under the first node. To analyze the behavior that SOAP causes, we can zoom in, to explore a subdiagram.

3. THE DIARY OF A DATE

We now explore the structure of the first step of the diagram shown in Figure 1. This example illustrates how to apply the structuring approach, and also shows the kinds of complexity that we have seen in real-world framework-based applications. We chose a benchmark that has been well-tuned at the application level to demonstrate the challenges of achieving good performance in framework-based applications.

We present an additional three levels of diagram. Two are the result of splitting according to architectural units (SOAP and the standard Java library), and one according to granularity.

Diagram level 1. Figure 2 shows the field-granularity activity that SOAP is responsible for, within the first transformation of Figure 1. The purchase date field flows along the middle row of nodes. Just at this level, the input bytes undergo seven transformations before exiting as a Calendar field in the Java business object.

The first transformation extracts the bytes representing the purchase date from the XML text of a SOAP message, and converts it to a String. The String is passed to a deserializer for parsing. The SOAP framework allows registration of deserializers for datatypes that can appear in messages. In the lower left corner is a sequence of transformations that look up the appropriate deserializer given the field name.

We highlight as a group the five transformations related to parsing, to make it easier to see this functional relationship. The first step takes the String, extracts and parses the time zone and milliseconds, and copies the remaining characters into a new String. The reformatted date String is then passed to the SimpleDateFormat library class for parsing. This is an expensive step, creating 39 objects (38 of them temporaries). Below, we

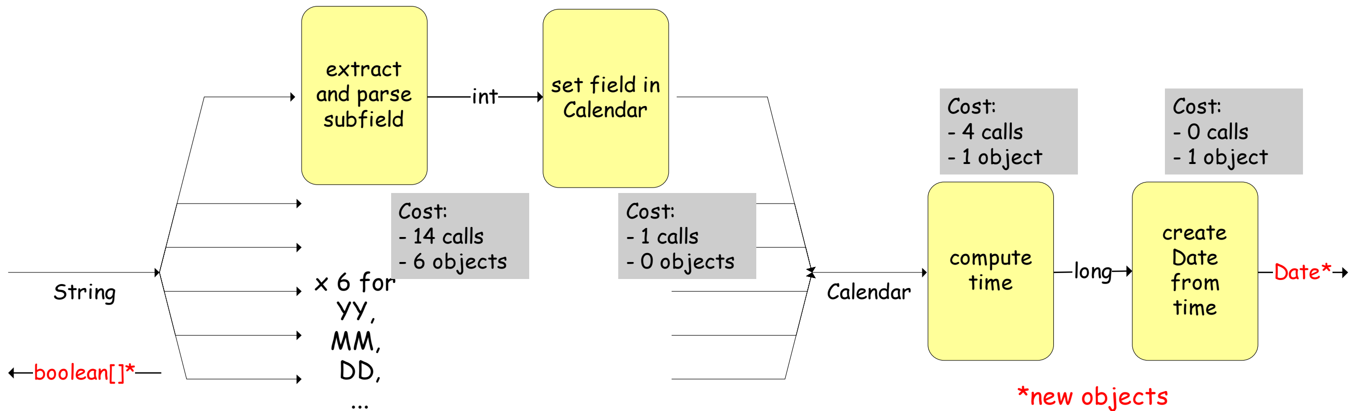


Figure 3: Further zooming in on the "parse using SimpleDateFormat" step of Figure 2 shows how the standard Java library's date-handling code transforms the purchase date field.

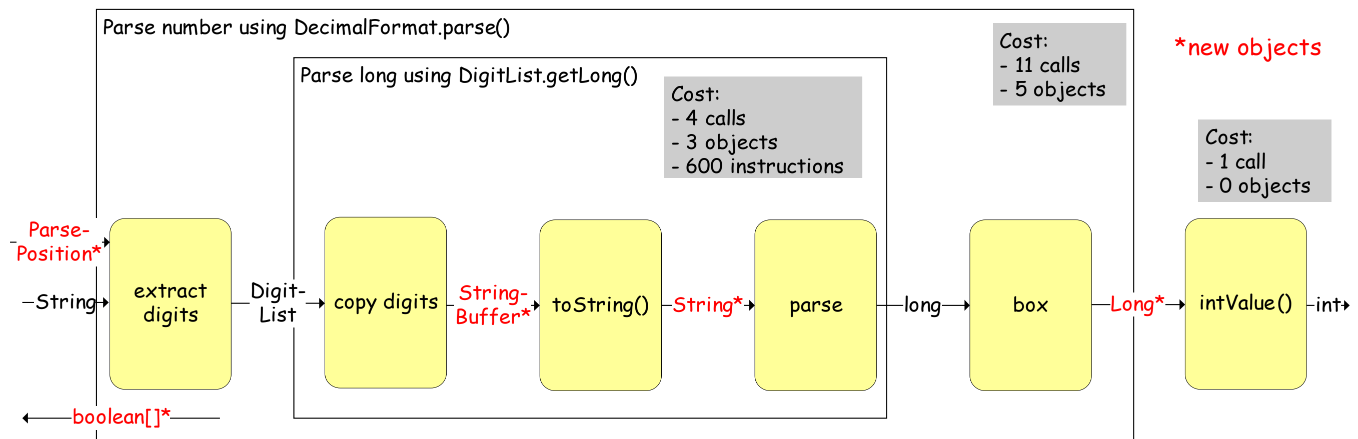


Figure 4: Zooming into the first step of Figure 3 shows how the standard Java library's number-handling code transforms a subfield of a purchase date (such as a year, month, or day).

explore the diagram, to find out why.[3] It then returns a new Date object, and joins that object with the original time zone and milliseconds.

The Java library has two date classes. A Date object stores the number of milliseconds since a fixed point in time. A Calendar stores a date in two different forms, and can convert between them. One form is the same as in Date; the other is seventeen integer fields that are useful for operating on dates, such as year, month, day, hour, or day of the week.

In the top row is an expensive transformation that builds a new default Calendar from the current time. Our Date object is then used to set the value of this Calendar again. Finally, that Calendar becomes the purchase date field of our business object, via a reflective call to a setter method. Java's reflection interface requires the Calendar to first be packaged into an object array.

Diagram level 2. Figure 3 zooms in to show the Java library's responsibility for the SimpleDateFormat parse transformation. The String containing the date is input, and each of its six subfields – year, month, day, hour, minute, and second – is extracted and parsed individually.

The SimpleDateFormat maintains its own Calendar, different from the one discussed earlier at the SOAP level. Once a subfield of date has been extracted and parsed into an integer, the corresponding field of the Calendar is set. After all six subfields are set, the Calendar converts this field representation into a time representation. This is then used to create a new Date object.

Diagram level 3. Figure 4 shows the detail of extracting and parsing a single date subfield, in this case, a year. Even at this microscopic level, the standard Java library requires six transformations to convert a few characters in the String (in "YYYY" representation) into the integer form of the year.

The first five transformations come from the general purpose DecimalFormat class. It can parse or format any kind of decimal number. SimpleDateFormat, however, uses it for a special case, to parse integer months, days, and years. The first, fifth, and sixth transformations are necessary only because of this overgenerality. The first transformation looks for a decimal point, an E for scientific notation, and rewraps the characters.[4] Furthermore, since DecimalFormat.parse() returns either a double or long value, the fifth transformation is needed to box the return value into an Object, and the sixth transformation is only necessary to unbox it.

[3] It often seems that things named "Simple" are expensive.

[4] It checks fitsIntoLong() on a number representing a month!

4. TRANFORMATION-BASED METRICS

Measures of the topology of a data flow diagram can give us some clues as to the complexity of an implementation. We can derive various measures from a single level of diagram, such as the total number of transformations and the maximum path length. For example, the first top-level step of converting a date to a business object field in Figure 1 is implemented by a total of ten transformations at the next level down – a sign that this is not a simple operation.

Other useful measures of complexity can be derived by looking at the entire hierarchy of data flow diagrams underlying a given transformation. These can give a sense of how "far afield" an implementation has gone from its high-level interface. Our top-level transformation hides three levels of detail, and takes 58 transformations in total. There are a total of 8 transformations at the first level of depth, 14 at the second, and 36 at the third. This breakdown shows us that much of the activity is delegated to a distant layer.

As we have seen throughout Section 3, structuring activity by transformations allows us to associate resource costs with transformations, rather than with program artifacts as is the case in traditional performance analysis. This has two advantages. First, it maps costs more closely to operations which may involve multiple methods or fragments of methods. Second, it enables comparisons across diverse implementations of the same functionality.

5. RELATED WORK

Recent work on mining jungloids [12] addresses a similar problem to ours, but at development time. They observe that, in framework-based applications, the coding process is difficult, due to the need to navigate long chains of framework calls.

There are many measures of code complexity and ways to normalize them, such as function points analysis [13], cyclomatic complexity [14], and the maintainability index [19]. Our measures are geared toward evaluating runtime behavior, especially as it relates to surfacing obstacles to good performance.

Performance understanding tools assign measurements to the artifacts of a specific application or framework [1,2,3,7,8,11,18]. Some have identified that static classes do not capture the dynamic behavior of objects [3,11].

There is much work on using data flow diagrams, at design time, to capture the flow of information through processes at a conceptual level [6,9]. In contrast, compilers and some tools analyze the data flow of program variables in source code [17]. In our work we use the data flow of logical content to structure runtime artifacts. This also sets us apart from existing performance tools, which typically organize activity based on control flow.

Finally, there is much work on recovering the design of complex applications [4,15].

6. CONCLUSIONS AND DIRECTIONS

That developers make such reuse of frameworks has been a boon for the development of large-scale applications. The flip side seems to be complex and poorly-performing programs. Developers can not make informed design decisions because costs are hidden from them. Moreover, framework designers can not predict the usage of their components. They must either design overly general frameworks, or ones specialized for use cases about which they can only guess.

We believe that elements of forming diagrams and grouping can be automated, for example, by using escape analysis, data flow analysis that combines static and dynamic information, and clustering based on descriptive labels (e.g. ones that identify data structures as records or fields) and application/framework boundaries. Programmers and designers must however remain a critical part of this process. Automation will also enable validation of the approach against a larger set of applications.

7. ACKNOWLEDGMENTS

We wish to thank Tim Klinger, Edith Schonberg, and Kavitha Srinivas for their technical contributions and support of this work.

8. REFERENCES

[1] W. P. Alexander, R. F. Berry, F. E. Levine, and R. J. Urquhart, A Unifying Approach To Performance Analysis in the Java Environment, *IBM Systems Journal* Volume 39 Number 1, 2000.

[2] G. Ammons, J. Choi, M. Gupta, and N. Swamy. Finding and Removing Performance Bottlenecks in Large Systems. *ECOOP*, 2004.

[3] E. Arisholm. Dynamic Coupling Measures for Object-Oriented Software. *Symposium on Software Metrics,* 2002.

[4] B. Bellay and H. Gall. An Evaluation of Reverse Engineering Tool Capabilities. *Journal of Software Maintenance:Research and Practice* Volume 10, 1998.

[5] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1, W3C Note 08, 2000.

[6] P. Coad and E. Yourdon. O*bject-Oriented Analysis*, 2nd Edition, Englewood Cliffs, NJ: Prentice-Hall, 1991.

[7] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by Analysis of Running Programs. *Workshop on Software Visualization, ICSE*, 2001.

[8] B. Dufour, K. Driesen, L. J. Hendren, C. Verbrugge. Dynamic Metrics for Java. *OOPSLA* 2003: 149-168.

[9] C. Gane and T. Sarson. *Structured Systems Analysis*. Englewood Cliffs, NJ.: Prentice-Hall, 1979.

[10] IBM Trade Web Application Benchmark http://www.ibm.com/software/webservers/appserv/wpbs_download.html

[11] V. Kuncak, P. Lam, and M. Rinard. Role Analysis. *POPL*, 2002.

[12] D. Mandelin, L. Xiu, R. Bodik, and D. Kimmelman. Mining Jungloids: Helping to Navigate the API Jungle. *PLDI*, 2005.

[13] J. J. Marciniak. ed. *Encyclopedia of Software Engineering*, 518-524. John Wiley & Sons, 1994.

[14] T. J. McCabe and A. H. Watson. Software Complexity. *Crosstalk, Journal of Defense Software Engineering* 7, 12.

[15] T. Richner and S. Ducasse. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. *ICSM*, 2002.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. *ASPLOS,* 2002.

[17] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 1995

[18] Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard. Efficient Mapping of Software System Traces to Architectural Views. In *CASCON*, 2000.

[19] K. D. Welker and P. W. Oman. Software Maintainability Metrics Models in Practice. *Crosstalk, Journal of Defense Software Engineering* 8, 11: 19–23.

A Model for Software Libraries

John M. Hunt
Clemson University
201 McAdams Hall
Clemson, SC
hunt2@cs.clemson.edu

John D. McGregor
Clemson University
312 McAdams Hall
Clemson, SC
johnmc@cs.clemson.edu

Abstract

Software libraries have long been an integral element of software development. Recent advances in areas such as software product lines and extensibility mechanisms have focused renewed attention on collections, particularly heterogeneous collections, of software artifacts. The contribution of this paper is to propose a model for a *software library*. Our work creates a framework that is abstract enough to encompass many kinds of software libraries beyond those used for the sort of programming constructs normally considered. This allows quite disparate collections to be understood within the same framework. Notable to our work is a discussion of the role of context and deployment to libraries. A comparison of our model to existing models is provided. A number of different types of libraries are analyzed to demonstrate the power of our model and to show how it leads to better understanding of several types of software collections.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Software libraries; D.2.13 [**Reusable Software**]: Reusable Libraries; D.2.13 [**Reusable Software**]: Domain Engineering; D.2.1 [**Re-quirements \ Specifications**]: Methodologies

General Terms

Design, Standardization

Keywords

Modeling

1 Introduction

Software libraries have long been an integral element of software development. Recent advances in areas such as software product lines and extensibility mechanisms have focused renewed attention on collections, particularly heterogeneous collections, of software artifacts. The contribution of this paper is to propose a model for a *software library*, briefly contrast our model to existing models, and show how it leads to better understanding of several types of software collections including: Dia Shape Sets, Eclipse plug-ins and software product line asset bases.

This model creates a framework that is abstract enough to encompass many kinds of software libraries beyond those used for the sort of programming constructs normally considered as libraries. The model allows quite disparate collections to be understood within the same framework. Notable to our work is a discussion of the role of context and deployment to libraries. A comparison of our model to existing models will be provided. A number of different types of libraries are analyzed including Dia Shape Sets, the Java Swing Library, Eclipse plug-ins and software product line asset bases.

2 The Importance of Models

The development of a standard vocabulary, requirements, and supporting models is an important step in the maturity of a discipline. Having the common understanding of an area that a model can foster has many advantages including:

- Improved ability to discuss problems and solutions
- A common understanding of the available design space
- An ability to compare solutions and techniques
- Guidance for those new to an area

The OSI seven layer model of a computer network [11] is a classic example of how a model can support the evolution of an area.

A better understanding of a domain, in this case the domain of libraries, should allow development of better products. For example, our model explains the connection between a library and a number of issues including how its assets are deployed onto production systems and the relationship between deployment issues and an asset‘s binding times. This is an area that is frequently ignored in library development, but may greatly effect the usefulness of the library. By providing advice about these issues we enable the development of better libraries.

3 Current Usage

While libraries are referred to frequently in the literature, a comprehensive definition has been lacking. Most writers are content to use the formula: ”A library is a collection of X.” Where X could be almost anything: functions, classes, architectures, use cases, test cases, documentation, specifications, or other artifacts. Examining how the term library is currently used, we find two different perspectives on library use:

1. A collection of software artifacts used by a developer, who is normally in a different organization from the library creators, to assist in the development of a program. Here the key problem is how someone unfamiliar with the contents of the collection finds and selects useful items. This often assumes the need to adapt the items found. The actual mechanisms, by

which products are composed, are not generally discussed.

2. A mechanism that holds a collection of artifacts for the purpose of facilitating composition with a product. The composition mechanism is often defined by the operating software or an intermediate runtime environment. Dynamically linked libraries (DLL) are an example. In this case, it is assumed that the desired items can be located and adaptation is not needed. The library users' problem of understanding and selecting assets is ignored.

While not contradictory, these two different uses of the term point to different aspects of libraries, both of which must be considered to gain a complete understanding.

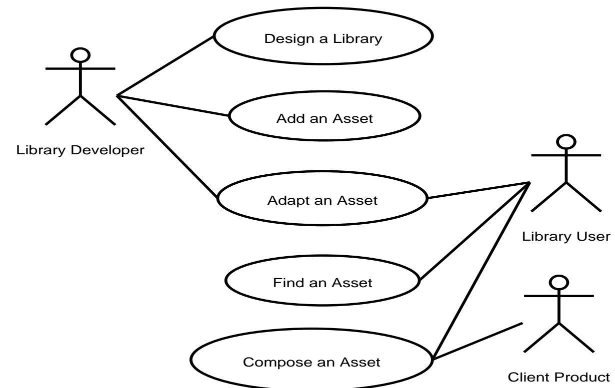


Figure 1. Use Case Diagram for Library.

4 Why Call It a Library?

The general notion of a library has several characteristics that apply to software libraries including:

- Library refers to both a collection of items and a facility in which to house the collection.
- Libraries typically organize their collection in a systematic manner and may limit their collection to have a common focus.
- Items are gathered together into a library to improve access and management.
- The library makes the items more public or available.
- The container for the items, be it a book case or building, improves access to the items.
- The library differs from a storage warehouse in that items are intended to be accessed frequently and individually.
- The library differs from a repository by making items more public, where as a repository removes items from circulation, making the stored items more protected, and in the process more private.
- Modern libraries are collections of many types of elements such as books, videos, computer programs, and many other elements.

The goal of a library is improved access and use of the items in its collection.

Software libraries have the additional constraint that we create them for the purpose of helping to develop products. As a result, while items in the general case are typically free standing assets, most assets in a software library will be parts or modules that acquire their ultimate usefulness only when combined with other assets, either from another library or custom assets, to form a product.

With this background, our basic definition for a software library can be stated as:

- A collection of composable assets,
- that contribute to building a product,
- aggregated within a mechanism that holds the collection for the purpose of promoting reuse by providing greater accessibility,
- for use by others.

5 A Software Library Model

In this section we present the results of a domain analysis of a *software library*. We use this method to present several in-depth discussions of the concepts found in the domain. We use the Unified Modeling Language (UML) to describe the results of the analysis.

The model we present is intended to be abstract enough to cover a very broad range of libraries, not just those containing programming language artifacts. Take, as an example, a drawing program that lets its user build a complex form and then store it into a palette for future use in other drawings. Such a palette of objects, when implemented in software, can be considered a software library. We want to be careful that such an example can be described by our model. Once a model of this generality is established, it can then be specialized for more common examples, such as programming libraries, or even further specialized, perhaps for class libraries. We defer these more specialized cases to future work.

5.1 Use Cases

We begin with the use case diagram of the domain, shown in Figure 1

The domain has three actors:

- The Library Developer provides the contents of the library.
- The Library User creates or maintains a product and composes the library's assets into the product.
- The Client Product is the product composed using the library assets. For some composition mechanisms the product acts directly to compose itself with the library. For example, resolving and executing a branch to use a shared library. In other cases, the client product is passively assembled by the library user. Even in these passive cases, the assets in the library must conform to the mechanisms used to compose the product.

The diagram has the following high level use cases:

- Design a Library - Since we believe that a library has more coherence then simply a collection of assets it should be designed as such. One obvious design activity is scoping, as a particular library should focus on a set of related abstractions. Library design also establishes the context in which the assets are intended to be used. Libraries should be designed so that

they are: complete, consistent, easy to use, and efficient [8]. A key to designing a library is understanding how it is used and how the division of roles between the library developer and the library user effects the design. This paper will focus on these use issues, rather than the design issues of a library.

- Add an Asset - The library developer creates the library by adding one asset at a time. Left implied is the ability to remove and modify assets already in the library. Adding an asset is singled out from other development activities as the result is visible outside the development environment.
- Adapt an Asset - By adaptation we mean the process of manually modifying a pre-existing asset for a new use. In the case of code, this is also referred to as code scavenging [9]. Adaption may be applied by a developer, or a library user in the case that the library user has access to modifiable asset, typically source. Adaptation may be applied to any pre-existing modifiable asset, not just library assets. For example, code examples from a text book could serve as the source for adaptation. While library assets may be adapted there does not seem to be any unique role that the library plays in this process that distinguishes it from other asset sources, as such it is not considered further.
- Find an Asset - The library user must be able to find assets that are appropriate to his problem. The user must then be able to understand the asset in order to determine whether the correct asset has been found.
- Compose an Asset - The library user must be able to compose assets into products. The ability to compose an asset into a software product in an automated fashion is what distinguishes a library from other collection of software artifacts. Assets in the library must be designed to support composition. It should be noted that some composition mechanisms are more flexible than others. The C++ template mechanism allows the related code to be composed with a variety of variable types. This sort of flexibility, does not use manual intervention, does not change the original asset, and does not add additional assets into the library. These difference distinguish this sort of flexibility from adaptation.

Conventionally, we think of the library user as being a product developer, who selects particular pieces of a library to be composed into the product. In this case, the ability to search within the library is important. For products that have an open extension capability, the end user who is engaged in product composition, is the library user. For example, the end user might modify a web browser by composing it with a plug-in. In this case, the user typically does not search and select from among library assets, but instead decides whether to compose a feature.

Allowing a user to compose features has the effect of creating a domain specific language, a language that corresponds to the problem domain and does not require understanding of or access to the solution domain. This is an important distinction from adaptation, which requires access to and understand of the solution domain, and does not provide a new problem solving vocabulary.

5.2 Concepts

The concepts needed to describe a library and its assets are shown in a class diagram in Figure 2. We provide a glossary in Table 1 to give a brief definition of the classes in the diagram. The key abstractions in the diagram are assets of various types. Of particular note are two types of composite assets whose main purpose is to collect assets - the library and the deployment container.

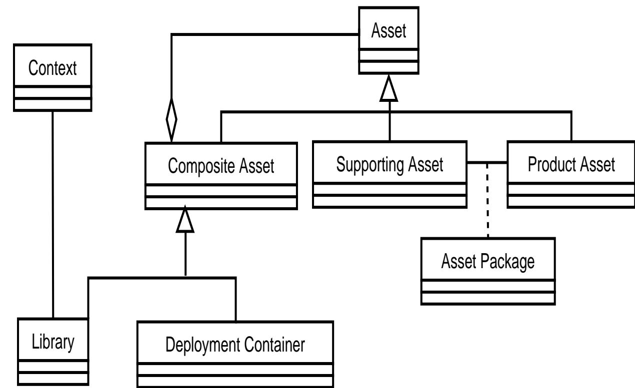


Figure 2. Top Level View of Library.

5.2.1 Relationships between Assets

The dominant relationship among the assets, at a conceptual level, is described by the composite design pattern [6]. This recognizes that one relationship among assets is hierarchical, i.e., assets may be composed of other assets. This has different implications for the different asset types. In the case of simple assets - product and supporting - this means that existing assets in the library may be used to compose new library assets.

For deployment containers the composite pattern provides three types of relationships:

1. A deployment container could contain a collection of simple assets for which it provides a composition mechanism. This is the most common case.
2. A deployment container could contain other deployment containers as one way to supply additional composition mechanisms. An example is an Eclipse plug-in which uses a jar file to hold executable assets.
3. A deployment container could contain a library. This would provide a way to move the library as a unit to other development systems.

For a library the composite pattern provides three types of relationships:

1. A library contains a collection of all the simple assets, this is the reason we have a library.
2. A library contains a collection of deployment containers. Having more than one type of deployment container allows the library to offer more than one type of composition mechanism to client products.
3. A library contains a group of related libraries. The context of a contained library must inherit the context of a containing library. A library may have more than one parent library.

5.2.2 Assets

The items we collect in the library are assets. We call them assets because we assume that they have value or we would not bother to collect them. We divide assets into product and supporting assets. Product assets are composed into a product. A supporting asset helps us make use of a product asset. For example, a Javadoc page

Table 1. Glossary

Term	Definition
Asset	an item that has enough value for us to collect. Three types of assets - product, supporting, composite
Product Asset	an item that is composed into a product
Supporting Asset	an item that supports use of a product asset, such as documentation
Composite Asset	a collection of assets; Types of composite assets - library, deployment container
Asset Package	shows the relationship between a product asset and its supporting assets
Library	collects assets and deployment containers
Deployment Containers	collects product assets in a way that is composable with a client product

for the Swing GUI is a supporting asset for the Swing class library. A product asset is not necessarily an executable asset. A help file shipped with the product is an example of a non-executable product asset. A build script is an example of a supporting asset that is executable.

What distinguishes a software library from other collections of software assets is the ability to compose library assets with a product, identified in the compose an asset use case. The importance of supporting this use case is what motivates the distinction between product and supporting assets. The asset package relationship groups supporting assets with the product asset with which they assist. If an asset package does not include a product asset, it cannot be composed into a product.

While executable modules, such as program functions, are the oldest and most widely used asset type, every phase of software development can take advantage of reusable assets, and a library can assist in increasing the reuse those assets. During the specification phase, we might use a library that includes standard use cases for some category of product to compose the use cases for our product. During the high-level design phase, we might use a library that includes UML diagrams describing standard subsystems to compose the product architecture. During the detailed design phase, we might use a library that includes pattern languages to guide the completion of the design. During the implementation phase, a library that includes code fragments might be used by a program generator or an aspect weaver.

In most existing libraries, the product assets of a library tend to be of the same type or at least apply to the same development phase. However, the assets in a library do not need to be homogeneous. We can gain considerable power by including all of the assets needed to produce a particular product. For example, a library might include assets, such as UML diagrams, that can be used in the design phase along with the executable assets needed for that product.

The assets we set out to collect are product assets, those assets that become or produce part of a product. The most common example is a source file in some high level language that compiles into a linkable executable. However, other inputs to a build process, such as frames, meta-models, layers, etc. may play the same role. As might a collection of predefined shapes for a drawing program. The supporting assets are collected to assist with using a product asset. If the product asset is removed from the library, the supporting assets should be removed as well.

The idea that assets are collected to build products means the ability to use an asset in multiple products is a planned result. In this view there is no such thing as a truly "general purpose" asset, that can be used in every setting [10]. We build a particular thing and the thing we are building imposes requirements on its parts [2]. The possibility of meeting these requirements by chance are quite low, this rules out several approaches to acquiring assets that have been commonly used.

5.2.3 Library

The term library refers not only to the contents of the asset collection but also the mechanism used to collect and manage the assets. The library has several attributes that are unique. The library is the level of abstraction where all three of the users we identified (library developer, library user, client product) are addressed.

To assist library users, library developers provide a number of supporting assets to explain and guide using the library. These library level aids might include such things as tutorials on the use of the library and example programs that show how library assets might be used to solve a common problem. The library may contain one or more search mechanisms, whose primary purpose is to assist the library user in finding assets. To support client programs, libraries should make product assets available in a composable way, often through deployment containers.

The library should add value beyond the value of the contained assets. The services provided by a library include:

- Collection support. Allow assets to be used as a group or individually. For example, copy or move an entire library instead of each of the contained items.
- Access or composition support. Assets are intended for use by client programs outside of the library. It should be possible to compose an asset in a client program without copying it out of the library. The library may provide multiple composition mechanisms that support different binding mechanisms.
- Selection support. A recognized truism in reuse is that an asset must be found before it can be reused [9]. This problem is more obvious in libraries since library users are a separate group from library developers. The library should provide support to the user to find assets.

5.2.4 Deployment Containers

Deployment containers allow the library's product assets to be composed with client programs without access to the library's development environment. Independently deploying a subset of the library assets in a composable way is the key to a library's ability to share and make public its assets, in contrast to other deployment constructs, such as repositories[1]. It is so fundamental to the library that the deployment container is often confused with the library. As can be seen from the use of the term "dynamic linked library" for what is actually only a deployment mechanism.

We can divide deployment containers into two categories:

1. Those that deploy library assets to product development systems. An example is the statically linked library, which is

[1] Recent literature often uses repository as a synonym for library, but typically ignores the aspect of deploying assets away from the development system. It is not clear if this difference is meant to distinguish the two.

composed during the development phase by a linker. Deployment to a product development system includes providing those supporting assets that assist the product developer.

2. Those that deploy library assets to production systems. Examples include the DLL (dynamic linked library) which the client links to at runtime. Putting deployment containers on the production system has the advantage that a single copy of the library assets can be used by multiple client programs. For example, most operating systems provide only a single I/O library for all of the hosted applications.

The composition mechanism supported by a deployment container determines the point in the software development process at which composition with the client program takes place; this is known as binding time. Libraries are usually assumed to have a single binding time, but this does not have to be the case. A library may have multiple deployment containers, each supporting a different binding time.

Once dispatched from the library development system, the deployment container and its enclosed assets are no longer under the management of the source control system. Therefore, deployment containers need a method of versioning independent from that of other development assets.

5.2.5 Context

All software artifacts are used in a particular setting or environment[10]. In this model, context represents the environment in which we intend to use the library assets. Context is included in our model to support the compose asset use case. As Alexander discussed in his classic teakettle example, the correctness of an artifact can only be understood in relation to the context into which we expect them to fit [1]. Alexander points out the dimensions in which an artifact must fit its environment are not enumerable, so a complete model of context is not possible. We show the major areas to be considered and discuss how context affects product development. Figure 3 shows our view of context as it applies to libraries.

We divide context into two parts [5]:

- the product domain, which describes what we are trying to build
- the solution domain, which describes how we can build a product

We further divide the solution domain into two parts:

- the platform, which specifies the library's dependences
- the architecture, which defines how the assets may be composed

If our library is contained within another library, its context includes that of its containing library. The contained library may impose additional requirements but must continue to meet all the requirements imposed by the containing library.

We associate the context with the library rather than the individual assets. This differs from other proposals, notably the OMG Reusable Asset Specification (RAS), discussed in section 5.2. There are several advantages to our approach:

- Context can be used to group assets. For a given project, we are typically only interested in a particular context. For example, a particular hardware platform or language may have been chosen for the project. Placing the context at the library level allows the entire collection to be evaluated, at least at this course level of decomposition, for suitability to the project without examining each asset.
- One of the barriers that prevents a library user from using preexisting assets is the difficulty in understanding those assets. This difficulty can be considered an additional cost of using the library, which in turn may cause the library user to choose to develop a new purpose built asset as a substitute for a preexisting asset from the library. Placing context at the library level allows the cognitive effort of understanding an asset to be reused, at least in part, over the other assets in the library. Thus, lowering the average cost of using a library asset.
- A shared context makes it more likely that assets from the same library are compatible. If there is no common context the inter-operating components may place different interpretations on data values leading to incorrect results. An example, this occurred recently when one component in the Mars Climate Orbiter pro-ject used english units and another component used metric units, resulting in the loss of the space craft. The loss of the first Ariane 5 rocket was due to a component that expected a different size for a numeric parameter then what was provided by another component. We will often want to use more than one asset from a library in the same product, such that the output from one asset will become input to another. An example where many components from the same library typically inter-operate can been seen with GUI libraries. Think of the difficulty if each widget used a different unit of size (pixel, point, pica, inch, centimeter, etc.) and a different coordinate system for positioning.
- Many characteristics of a good library [8] depend on the assets in a library exhibiting similar behavior and usage characteristics. This similarity can most easily be achieved by placing context once at the library level, rather than trying to insure that contexts for each asset have the same values. Examples of such characteristics are: consistency, ease-of-learning and ease-of-use.

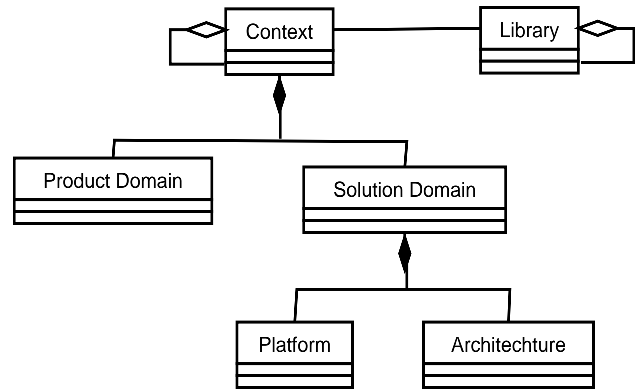


Figure 3. Context Detail.

6 Contrast with Other Models

6.1 IEEE Standard 1420.1

Despite the longevity of the software libraries the only previous attempt we were able to find to provide a model was made by the Reuse Library Interoperability Group (RIG), an industry consortium [3]. Their work was published as IEEE standard 1420.1[7]. Their goal was to provide only an interface definition to exchange libraries, not a complete model. Much of this work is related to tracking software certification and specifying intellectual property rights, which was codified in standards 1420.1a and 1420.1b respectively. These issues are not of interest to us and will not be considered further.

The model provided includes only a class diagram; there is no use case diagram. This makes it difficult to be sure what they see as the overall role of the library. The existence of pre-existing libraries in the model implies the role of library developers. The use case they explicitly support is selecting an asset for reuse. There is nothing in the model to specifically support the composition of assets with client programs.

In the class model there is no equivalent to deployment container, which selects and supports assets for composition, this also fails to support a way to specify binding times. The library class lacks any attributes to assist in selection or limit searching without looking at all the assets contained. There is no way to specify a constraint on the asset to be included in a library. This is somewhat surprising as members of the RIG group stated that they believed libraries should be focused, specialized, collections, not general purpose[3]. Moving the domain attribute from the asset to the library class would help here. The role of architecture in reuse is completely ignored. There is no distinction between assets that are used in the product and supporting assets. In short, the role of context is ignored.

6.2 OMG Reusable Asset Specification (RAS)

OMG Reusable Asset Specification (RAS) [12] primarily models assets, however, it also models the relationships between assets, and even (briefly) discusses a repository for assets. In this model, the asset, which is often referred to as an *asset package* in the standard is composed of 5 parts: solution, profile, usage, classification, and related asset. The solution asset corresponds to our product asset. The other parts make it easier to work with the solution, which corresponds to our supporting assets.

While we provide a descriptive model, RAS is a proscriptive approach, which imposes requirements related to OMG's Model Driven Architecture (MDA). It is interesting that to provide automated assembly in MDA a large amount of context information is required. RAS puts its context information at the asset level. As has been noted, this means all use decisions must be made for each asset, instead of reusing information about the collection. In practice, this may be mitigated by a combination of the large granularity of the components intended for RAS and the planned development of detailed implementation profiles.

The RAS standard also defines RAS Repository Services. These define Java and HTTP methods to store and retrieve assets from a RAS Repository. However, no model is provided for the repository. The RAS glossary defines a repository as: "A centralized access and storage point for reusable assets." It is not clear from this which use cases are envisioned. It does not seem to involve the actual composition of assets. The text mentions that these services are intended for small and medium repositories. However, it is not clear how the "centralized access and storage" is related to the multiple repositories. Also, there is no guidance on why an asset would be in a particular repository or how assets in a particular repository are related. The glossary also defines a reusable asset library as a"conceptual composite artifact that encompasses all possible reusable assets" which sounds much like the failed general purpose library paradigm. This concept of library is not otherwise referred to in the document and does not explain its relationship to reliable asset repositories.

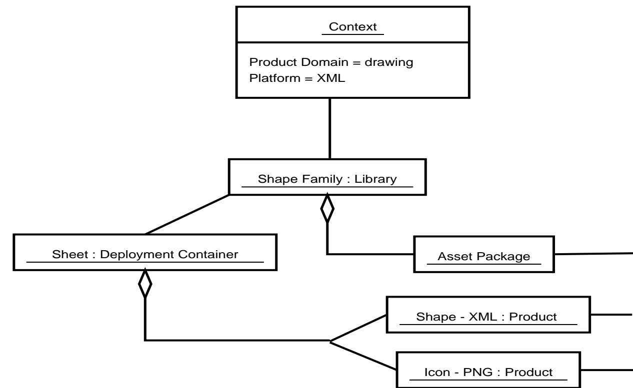


Figure 4. Dia Object Diagram.

7 Analyzing Some Examples

To validate our library model we analyzed a variety of libraries, to check if the model can describe them. Here we present as examples: Dia Shape Templates, the Java Swing Library, Eclipse plug-ins and Software Product Line asset bases. Swing will represent a typical class library. Eclipse and asset bases are not generally thought of as software libraries; however, they are collections that fit our library definition. Studying these examples can show how more extensive use can be made of libraries particularly in terms of improved asset composition. They illustrate the importance of context to library usage. They also provide an example of how a standardized model can help explain new material.

Both Eclipse plug-ins and Software Product Lines will show us something about the future directions of the software library domain. Domain choice drives the architecture and related design rules. Having a well defined context for product and solution domains supports the design of assets that are composable without modification.

7.1 Dia Shape Sets

The Dia drawing program allows users to define shape sets. The object diagram for Dia shapes is show in Figure 4. Dia is a drawing program designed to draw different types of diagrams. It comes with shape sets for drawing such things as electronic circuit diagrams, UML diagrams, flowcharts, etc. The main abstractions the program works with are shape objects and connectors.

The basic asset type in Dia is a shape. Dia supports the hierarchical composition of shapes. Existing shapes can be used to draw a new compound shape, which can be saved as a shape. Shapes can be collected into sheets. This allows related shapes to be collected

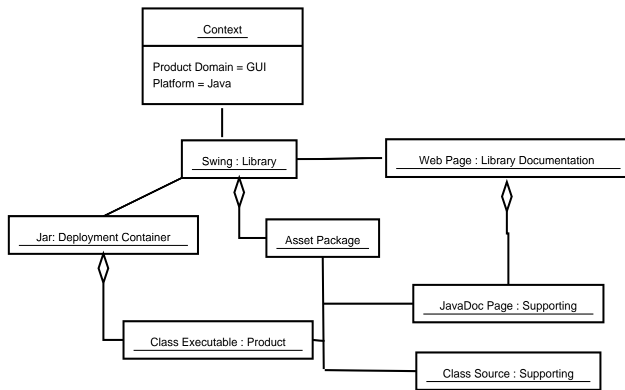


Figure 5. Swing Object Diagram.

together. For example, AND, or, NAND, and XOR shapes are collected in a circuit sheet. Sheets allow a set of shapes to be pulled into the program as a group and made readily available to the user. Adding a shape set extends the programs capabilities.

A typical asset package, for a shape, has two product assets, a file which provides an icon to represent the shape on the palette menu, and a file which has an XML description of the shape which the Dia program can translate into a drawing. A sheet acts as a deployment container, grouping a collection of shapes together and making them available for composition. In this case the library user is working interactively with the library, so the search / selection mechanism is integrated into the client program. The user selects a shape by choosing a sheet from an alphabetized list of names. Choosing a sheet causes the icons for the sheet's shapes to be displayed in a palette window. While the program comes with a large number of shapes, and allows new ones to be built, the ability to organize shapes into groups with the sheet mechanism keeps the number of shapes being worked with at a given time to a manageable level even with these simple search mechanisms.

Even though Dia shape sets are rather different from what is normally thought of as a software library, they meet both our understanding of a library, as well as the current usage of the term, that is a collection of assets. This object diagram shows that our model is able to accommodate them as well.

7.2 Java Swing Library

The Swing library is typical of Java class libraries. An object diagram for the Swing library is provided in Figure 5. It is similar in structure to many other class libraries, but provides a better documentation specification and supporting tools (such as Javadoc and Jar files) than most library systems.

A typical asset in a Java library is a class including the special comments that are used as input for Javadoc. Physically the class is defined in a single source file ending with a .java suffix. The class source code file is the input to the Java compiler to produce an class file and to the Javadoc tool which will produce a hyperlinked web page to document the class. The asset package for each class will bundle a Java source file, executable class file, and a web page.

The library provides documentation, a search mechanism, and a deployment container. Javadoc specifies a standardized format describing the library as whole. This page provides hypertext links to other library documentation, such as tutorials, and hypertext links to the generated web pages for each of the classes in the library. The links to the class web pages form the primary index for the library, based on an alphabetized list of class names. Since this documentation conforms to the standards for the web, any web-based search engine can provide an additional basis for search using text matching.

The Java language uses dynamic class loading rather than linking. While the executable class files can be used directly by a client program, Java also defines an archive file format, Jar, which serves as a deployment container. The Jar was designed to move a collection of class files around as a unit. Jar files have roles beyond deployment. A client program can load a class from a Jar file without unpacking it. If a program's main method is in the Jar file, it can be tagged to execute without unpacking the Jar file. Jar files can also include security information about a group of classes.

Much of Java's success is attributable to the support provided for libraries. Javadoc provides Java libraries with extensive, maintainable documentation that has a consistent look and feel and a consistent search mechanism. The Java virtual machine provides a hardware independent platform thus eliminating one major aspect of architectural mismatch. Many issues, such as memory management, normally left to applications are handled by the Java runtime environment, reducing the number of different context dependencies.

7.3 Eclipse plug-in

Eclipse is a modular, open-source product that provides an extensible Integrated Development Environment (IDE) [13]. Its goal is to provide a single user environment that supports the integration of development tools produced by different organizations. Eclipse allows the user to build a version of the product that fits their needs by installing appropriate modules. While Eclipse is very modular in many ways it is also designed to fit the needs of a specific domain (IDE), a specific platform (Java), and provides a specific architecture that modules must adhere to in order to be composed. An object diagram which presents Eclipse modules as a library is provided in Figure 6.

The typical Eclipse module provides a development tool or closely related tools. For example, to support a compiled language requires not only a compiler, but also a language specific editor, templates for the different file types in the language, debuggers, wizards, and a variety of documentation. Eclipse differs from most libraries by the emphasis it puts on supporting assets; and its support of open module composition after deployment.

Tools for Eclipse are deployed as features. A feature is a group of plug-ins that are deployed or upgraded together. For the user, the feature is the unit of both deployment and versioning. Physically a feature is composed into a compressed file, to allow it to be moved as a unit. Each plug-in in the feature has its own directory. Plug-ins provide or support different parts of a feature, as they extend different parts of the Eclipse platform. For a compiled language, we would expect the editor to be placed in one plug-in, while the compiler, which can be run without a user interface, would be in a different plug-in. Each plug-in must provide a specification, called the manifest, written in XML, that describes the plug-in to the Eclipse platform. Beyond the manifest, the assets provided for a plug-in vary. If a plug-in has an executable portion, it must consist of Java class files collected in a Jar file, located in the plug-in directory. This is an example of one deployment container, the plug-in, containing another, the Jar file. Documentation is stored in the plug-in

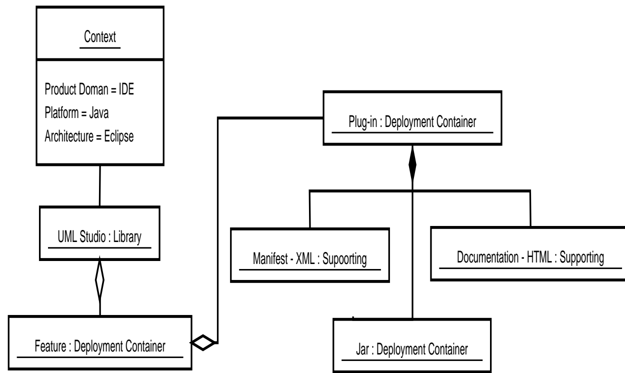


Figure 6. Eclipse Object Diagram.

directory as html files, and may be put into its own plug-in to make it easier to internationalize. Other assets found in plug-ins might include icons, images, web templates, etc.

The Eclipse model provides support for all three of our use cases. Adding assets is supported by a number of tools, such as the Plug-in Development Environment (PDE), which provides wizards to walk the asset developer through the process of adding assets. Searching and understanding assets is supported by search features and an expandable help system. Plug-in assets can specify how they should be included in the table of contents for the help system. A number of ways to compose assets are provided. Plug-ins can modify the behavior of other plug-ins by extending them in an inheritance relationship or can use other plug-ins by specifying a dependency.

7.4 Software Product Line Asset Bases

"A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [4] SPL scopes which products will be included in the product line early in the analysis. This allows the development of a common architecture which has explicit variation points identifying where and how different product within the product line will vary from each other.

The core assets are collected in an asset base to be used in developing the products included in the software product line. The asset base is a heterogeneous collection, which includes assets for all development phases, from requirements to implementation. It is tightly scoped to include only assets that will be used in more than one product in the SPL. A SPL may use library mechanisms to group and prepare implementation assets for composition with products. Thus, an SPL's core asset base is an example of a library containing libraries. While specific to a particular group of products, the size of an asset base may reach millions of lines of code.

The major differences between an SPL asset base and a traditional library can be seen in the context provided and the relationship between reusable asset and the products built with them. The SPL development process begins with a domain analysis which is further defined by a selection of features to be supported and a grouping of those features into products. So the product domain portion of the context is well defined for SPL. The product domain is further constrained by the selection of products that will be supported by the product line. In contrast, other than a tool domain, such as GUI development, we typically don't know the bounds of the product domain for a reuse library. Within the product domain a traditional library is intended to be used in an open set of products.

In the solution domain, an SPL will typically define a single architecture to be supported. A typical reuse library will attempt to support multiple and undetermined architectures, often aiming for the difficult goal of being architecturally neutral. SPLs may support multiple platforms, but the platforms supported are made explicit. The influence of platform variants can be shown in the variation points of the SPL's architecture. Reuse libraries typically support a single platform, but platform information is often implicit.

The relationship between the collected assets and the products produced also differs. SPL products are made primarily by assembling assets from the asset base. Ninety percent reuse levels are typical in product lines, with many reaching a hundred percent. All assets for a product line are collected in a single asset base. All of the assets in the asset base should be used in multiple products. Traditional reuse libraries support a much lower frequency of reuse, typically not exceeding fifty percent. Achieving this involves finding, understanding, and using many different reuse libraries. A given product will use only a small percentage of a library. It is possible that many library assets will never be used in a product.

These differences are summed up by Clements "Software product lines represent a significant departure from software re-use schemes in which attempts are made to make assets as general as possible without the context provided by an architecture and a scope definition, and from opportunistic reuse schemes in which low-payoff assets are scavenged ad hoc from a reuse repository." [4].

7.5 Summary

These brief examples show very different collections. The type and strength of relationships among the elements in the libraries are different. In the Eclipse example, the elements would be expected to be consistent with one another. On the other hand, a product line asset base may contain assets where chosing one asset excludes chosing another, an exclusive-or relationship. The Eclipse example has a very clear need for completeness - the plug-in needs to work - while the product line asset base may be quite incomplete. There are several directions in which this work needs to be extended:

1. The existing model should be applied to additional types of libraries. The examples in this paper, Dia shapes sets, Eclipse plugins, and software product line asset bases, in addition to the programming libraries normally considered, suggest the diversity of libraries that should be addressed.
2. The model is presented at a very abstract level to allow it to cover the maximum range of libraries. Specializations of the model should be developed for important categories of libraries, the most obvious being programming libraries. This more specific model could consider common programming issues such as error handling and memory management. A programming library model could be further specialized to handle common cases such as class libraries, active libraries, etc.

8 Conclusions

Software libraries are one of the oldest, most used approaches to software reuse. Despite their long past and interesting future, there has been almost no research on libraries as a product domain. Based on experiences with other product domains, a thorough domain

analysis advances the state of the practice.

We have presented an analysis of libraries as a product domain, beginning with the definition that a library is: A collection of composable assets, that contribute to building a product, aggregated within a mechanism that holds the collection for the purpose of promoting reuse by providing greater accessibility, for use by others. While this covers many different types of asset collections it excludes many as well. A contrast can be seen in the World Wide Web. The web is a collection of assets, including software, and it provides search mechanisms to assist in finding the desired asset. Yet, it is not a software library, because most of the assets are not intended to be composed into products and because the web does not provide a composition mechanism.

We have provided simple, but comprehensive, use cases. We have three actors (library developer, library user, and client program) and three essential use cases(add an asset, find an asset, and compose an asset). With these use cases we avoid focusing exclusively on either the search problem or the composition problem. As a result, we highlight the need to be able to compose the assets found into a product.

In our model by making the deployment container a separate entity and allowing multiple instances, we open the way for support of multi-binding time libraries. By making a clear provision for a multi-binding time library we provide the opportunity for better library support for product lines, where multiple binding times is a significant issue.

Finally, we have clarified the relationship between a library and its context. There is growing acceptance that reusable software requires an explicit context. This applies to libraries as well. The library context includes both product domain and the solution domain of both platform and architecture. Our model identifies the appropriate concept with which to associate context is the library, not the individual asset, as has been the case in other work. Placing context at the library level allows assets to be grouped by context, allows better reuse of developers understanding, and produces the situation where a library's assets are compatible with each other.

Libraries continue to be an important means of providing reusable software; however, they are still understood, designed and built in an ad-hoc manner. This paper by providing the top level requirements and model is intended to provide a starting point for a comprehensive approach to library development and use.

9 References

[1] C. Alexander. *Notes on the Synthesis of Form*. Harvard University Press, Cambridge, Massachusetts, 1964.

[2] C. Baldwin and K. Clark. *Design Rules: The Power of Modularity*. MIT Press, Cambridge, Massachusetts, 2000.

[3] S. V. Browne and J. W. Moore. Reuse library interoperability and the world wide web. In *Proceedings of the 19th International Conference Software Engineering*, pages 684–691. ACM Press(New York), May 17-23 1997.

[4] P. Clements and L. Northrop. *Software Product Lines, Practices and Patterns*. Addison-Wesley, Boston, Massachusetts, 2001.

[5] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1999.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1995.

[7] IEEE. Data model for reuse library interoperability: Basic interoperability data model (bidm). Standard 1420.1, IEEE, 1982. Standard for Information Technology - Software Reuse.

[8] T. Korson and J. McGregor. Technical criteria for the specification and evaluation of object-oriented libraries. *Software Engineering Journal*, 7(2):85–94, 1992.

[9] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[10] J. McGregor. Context. *Journal of Object Technology*, 4(7):35–44, September-October 2005.

[11] L. J. Miller. The iso reference model of open system interconnection: A first tutorial. In *Proceedings of the ACM '81 conference*, pages 283–288. ACM Press (New York), 1981.

[12] OMG. Reusable asset specification. Standard RAS, Object Management Group, 2005. http://www.omg.org/docs/ptc/04-06-06.pdf accessed July 14, 2005.

[13] OTI. Eclipse platform technical overview. White paper, Object Technology International, Inc., 2003. Paper http://www.eclipse.org/whitepapers/eclipse-overview.pdf accessed July 14, 2005.

Making a Boost Library

Robert Ramey
Software Developer
830 Cathedral Vista Lane
Santa Barbara, CA 93110
(805)569-3793
ramey@rrsd.com

ABSTRACT

Boost is a loose organization of C++ developers dedicated to the creation of high quality C++ libraries. It can be found at www.boost.org [1]. This article describes the process of getting a library accepted into Boost along with advice from one who has been there.

Categories and Subject Descriptors

D.2.13 [**Reusable Software**]:Library Software Development.

General Terms

Design, Human Factors, Legal Aspects.

Keywords

C++, Boost, Libraries.

1. WHAT IS BOOST?

Have you ever wanted to:

- do a really, really good job at something?
- provide the "definitive" or best solution to some problem?
- make something that lots of really smart people would appreciate and use?
- work to a higher standard than your current job requires or will permit?
- demonstrate that you are really a good programmer?

Maybe you want to consider making a library and submitting it to Boost.

www.boost.org is a loose organization of C++ developers dedicated to the creation of high quality C++ libraries.

Boost libraries are distinguished by:

- Wide applicability – libraries are usually things that are widely applicable. The effort required to write a library and get it accepted to Boost is not justified unless the library is going to be re-used many times. For this reason, many (though not all) Boost libraries are fundamental building blocks like smart_ptr,
- As a corollary to the above, Boost libraries are portable. They are written to the C++ language and library standards with work-arounds for bugs in specific compilers. Most Boost libraries leverage on idioms already in boost which already have been implemented in a portable way.
- Another corollary to the above is that Boost libraries tend to strive to be the "best" or "definitive" solution to a particular problem.
- There is relatively little repetition of functionality within Boost. If there is a best and/or definitive solution to a problem, other libraries generally incorporate it.
- Boost libraries often use cutting edge techniques such as template meta-programming to achieve desired goals.
- Boost libraries strive for high quality. This is attained via an exhaustive testing discipline and corresponding infrastructure.

The above common library features are the result of a very public, rigorous and iterative peer review process that draws on the experience and knowledge of the entire Boost community.

Boost libraries cover a wide range of functions and applications. Among the most wildly used are regular expression parsing (regex), smart pointers (smart_ptr), threading, date time, file system, preprocessor, testing and correctness and others. It is really not possible to convey in a short paragraph the breadth of these libraries. A complete list can be found at [4]

All Boost libraries are subject to the Boost License [2] which is designed to permit usage of the library as widely as possible.

As the author of the recently accepted Boost Serialization Library, I can attest that making a library and getting it accepted into Boost is much harder than it would first appear. This article describes Boost and what it takes to get a library accepted. Note that opinions and advice expressed here are my own. I do not presume to speak for any other Boost members.

2. THINKING ABOUT YOUR LIBRARY

We all have at least a few really great ideas.

2.1 Some Ideas Are Really, Really Hard to Implement.

Some things are inherently difficult. One recurring idea is a dimensional analysis and units library. C++ operator overloading makes this idea very appealing, so it is easy to get started. There are many libraries available in this domain and several have been submitted to Boost. None has yet reached the formal review stage. This may be because there is wide applicability of such a package and it is very hard to reach a consensus on requirements and implementation. Many people need dimensional analysis and

have made and used libraries that suit their needs. Making one library that covers enough applications may be just too difficult, so no consensus has been reached.

2.2 Consider Making a Smaller Library

As we will see below, making a Boost library and getting it accepted can be a huge undertaking. Before embarking on the process, you should consider if you can see it through. It may be a better choice to make a smaller library.

Many of the most useful and widely used libraries, such as STATIC_ASSERT, are small but tricky.

Even a small library will entail more work that you might think. Better to make something small that is really useful rather than something bigger that does not get finished.

2.3 Start Writing Documentation.

I know that seems backward to a lot of people, but bear with me.

- Description – what this library does.
- Motivation – why is such a library useful?
- List of features required by such a library.
- List of other libraries that do something similar and how your library is different and or better.
- To get started, you will have do some research. Be sure to include the Boost website in your search:
- Website. Something like your library may already be in Boost. Or perhaps something you can build on is already there.
- Mailing list. Here is all the information about previous proposals and submissions. It is quite possible that something similar to your library has been submitted in the past and not been accepted for some reason. If so, you need to know it. It is also possible that the problem your library is intended to solve has been discussed.
- Files section contains libraries that have not been formally accepted into Boost, for various reasons. Some may be in process of development. In many cases the library is more of an experiment than a full blown library. The author might have submitted the library but did not have the time to push it all the way through the process. In my view the files section is an underappreciated gold mine of useful code. I look through it all the time when I have a small sticky problem. Need to render in integer in roman numerals? It is in there!

Your library should leverage facilities already in Boost rather than re-invent any wheels so you can spend all your time concentrating on the unique aspects of your package.

At this point, Boost recommends that you query the list to see if there would be interest in your submission. This is commonly done. Personally I do not think such a query is always a great idea. If you have done your research, you should have a good idea whether or not your proposed library will be interesting. When you query the list you risk getting involved in an opinionated discussion that revolves around a still nebulous idea. My view is that the real issues do not present themselves until some code is written, tested and compiled. My (silent) reaction to such queries is: Hmmm – might be interesting, let's see the code and some documentation.

2.4 Become Familiar with Boost Tools

Start out by installing the current Boost libraries on your development system. Boosters think this is easy. And it is, after you are familiar with it. It means getting paths and environmental variables setup for the command line version of your favorite compiler and a couple of other things. Unless things go perfectly the first time, you will have to investigate how the build system works which takes some time. For this reason lots of users of Boost libraries just incorporate Boost source code headers into their projects. Many of the Boost libraries are supplied as header files and do not require the building of linking libraries. However, for a library developer you will have to become familiar with the whole system.

2.4.1 bjam (boost jam)

Boost has its own system from building executables and running tests. It might best be described as a next generation of UNIX make. The main component **bjam** processes a Jamfile which describes the requirements for buiding libraries, and executables. Dependencies between header and source files are handled automatically. Also compiler, library, and platform dependencies are also handled automatically. Generally, there is little or no compiler, library, or platform specific information in a Jamfile. In this way, your library will be built and tested on other platforms without anyone having to do anything special. Of course that is the theory. In practice there is usually a little bit of effort required to specify small adjustments required for different environments Without **bjam**, it would be a huge effort just to test someone else's code – now it is manageable.

bjam is used to build libraries and also run a test suite for each library. Information on using bjam is spread among several web pages on the boost site. Perhaps the easiest way to get familiar with bjam is to use the bjam files for other libraries as models for your own.

Unfortunately, it is one more thing to learn and at the beginning it will feel like its slowing you down. In fact, it IS slowing your down. But the investment in effort to become familiar with it will be paid back many fold as your library becomes more elaborate and ported to more platforms.

2.4.2 Documentation

As I write this, most of boost documentation is in HTML files. This is considered acceptable for new submissions. These files may be generated by hand or with another tool of your own choice. To save time I used a skeletal set of HTML files from boost that provided all the sections, and style information that is common to boost libraries. I found this very helpful. Boost is moving towards a new system for documentation, Boost.Book**,** which maybe worth investigating.

2.4.3 Boost Test

Fundamental to a library submission is a test suite. The key tool for building tests is the Boost Test library. This described in the Boost library documentation in the section "Correctness and Testing" [5]. Tests are run on separate test servers and produce a daily test matrix which shows all test failures organized by library and compiler.

2.4.4 *Other Boost Tools*

It is really necessary to have reviewed most of boost libraries to understand what is available already. Boost contains lots of code to simplify program portability, ensure correctness and implement commonly required idioms. Code that needlessly includes functionality already in boost will probably not be accepted. It takes a little time to become familiar with all this.

3. CRAFTING YOUR SUBMISSION

Now you are ready to write your code. More likely, you have got your original code and you are ready to start making it acceptable for Boost.

In order for your library to be evaluated, others will have to experiment with it, test it and use it.

If someone wants you to try out their code, they had better make it as easy as possible for you – correct? You could not spend time fiddling around with compiler settings, deciphering incomplete, or unclear documentation or otherwise wasting time. Well, surprise, no one else can either. Be prepared to submit a self contained package that "just works". Given that the B**oost** community uses a variety of compilers, libraries and platforms, this challenge might seem impossible at first glance. Boost tools provide a solution to this problem, so now you will start modification of your code to do it the "Boost way".

Code, make test, add to Jamfile, debug, add to document, redesign, re-factor and repeat until done. Soon you should have the following in your personal copy of the Boost directory tree:

- Code for your library's headers and source files.
- Code for tests and demos
- Jamfiles for build and test
- Documentation for library usage

When crafting your library:

- Work to the C++ standard – not to a particular compiler.
- When the compilers you use to test cannot handle the standard conforming code, make changes to achieve portability desired. Use facilities already in boost to achieve the desired portability.
- Use at several different compilers. This will increase the number of people that can/will tryout your library. All compilers have bugs and quirks. Building your code with more than one compiler/library helps make your code more portable and standard conforming.
- Leverage other Boost libraries to achieve portability and gain "free" functionality.
- Add a section to your documentation titled "Rationale". As writing on your library progresses, you will be required to make non-obvious design and implementation decisions. For each of these decisions, add an explanation to the "Rationale". Later, when it is asked why you did something a certain way, you will not waste a lot of time re-discovering your original reasons.
- Include a tutorial with an example program in your documentation. This should permit an interested party to see the utility and ease of use of the library in a very short time. In a sense, the function of this section is to "sell" the library to a potential user..
- Include reference documentation to catalogue its features and usage.
- For each library feature, include a test and add it your Jamfile. Also add an entry to your reference documentation.
- Repeat the cycle, adding features until the library is mature enough to demonstrate its utility, design, direction and final form. It does not have to be complete, but it should have functionality that others can benefit from, including working code, documentation and tests.

Eventually you should have enough of the library done that it can be evaluated. The documentation and code might have some placeholders but it will be clear what you envision as the final version. Now you are ready to submit to Boost for preliminary consideration. Announce your submission on the Boost developer's list and make it available to interested parties in one or more of the following ways:

- zip, gzip, or tar your submission into one file. Be sure to retain the Boost compatible directory structure. This can be uploaded to he Boost files section and/or to your personal website.
- Request CVS access to the boost-sandbox project and check it in there.

If all goes well, you should get some feedback on the list within a couple of days. If you do not get any feedback, try announcing again as sometimes the list is focused on a heated technical debate, formal review in progress, new release or something like that. The Boost community is large so usually there is someone interested in just about any topic posted.

Hopefully some people will find your package interesting, useful and transparent enough to experiment with.

4. DEALING WITH FEEDBACK

Now the fun begins. Hopefully you will get some feedback. Hopefully at least some of it will be positive. Here is what you might get.

4.1 It's Got Bugs

Well, shame on you. You did not test it exhaustively enough – this is a big turn off for users. Think of your own reaction when you have a problem, find something that purports to solve it, and it turns out to be more trouble than its worth. You are disappointed and reluctant to trust the library (and its author) again. Do not do this to your users or to yourself. Better to have something useful that has a path to the future than something that does not work along with a promise about how great it is going to be.

Maybe it is a misunderstanding – ask the user to run your tests – or add a new test.

4.2 It's Useful But It Needs Feature X

Now we are getting somewhere. Someone is actually trying to use it. This is a big accomplishment. Feature requests come in lots of flavors.

- There is a way to do it – it is just not clear from the documentation and examples. So it is just a misunderstanding. This is your cue to add another section to your documentation with a supporting example and test.
- It is already planned for the future. Very good – they want more. Acknowledge the request and put it your list.
- You never considered it, but it is a good idea. Great, put it on your list.
- You are convinced feature X is not a good idea. – explain why this feature is not included and not planned. This may start a discussion thread. If you already considered this and rejected it, it should already be in the "Rationale" section of your documentation. Eventually it will get sorted out and hopefully a consensus will be reached. Be sure that the entry in the Rationale reflects the relevant aspects of the discussion.

4.3 How Do I Use the Library To Do X?

This question is similar to a feature request. It will result in either a promise to add a new capability, an example or demo with an attached explanation showing that the capability is there, or a new entry in the documentation indicating why the capability is not there.

It is often easier to write a small demo showing how to do something than it is to explain how to do it. It is much better than a seemingly endless back and forth on the mailing list. Add the demo and explanation to your test set and documentation. If you do not do this, the same question will come up again and again.

4.4 Personal Comments

Boost mailing list discussions are governed by the Boost Discussion Policy [3]. This policy is designed to make mailing list discussion as productive as possible and avoid problems which can plague other mailing lists. Among other things it proscribes personal attacks and admonishes list members to stay on the subject. Members on the list use their real names. Occasional gentle reminders keep this list functioning in a product and professional way.

Of course someone might slip and say "This feature is useless" rather than "When would I use this feature?" which is far more likely to elicit a useful response. Should something like this happen do not take it personally. Often what seems offensive is a communication problem. Remember that the Boost list is a world wide community. Sometimes there can be misunderstanding because of language difficulties. Unless you are in a position to respond to a poster in his native language, have some patience. Generally I like a little humor in posts – but remember that it can be the source of a misunderstanding and someone can be offended when there was no such intention.

Often the poster may have a valid point even though he phrases it in an irksome manner. Just respond to the substantive point and ignore the rest.

When arguing issues related to your library, stick to the Boost discussion policy.

The most heated discussions revolve around differing opinions and hypothetical situations. Often time the best thing is to make a small test program and get some real facts. That will usually resolve things. In any event, the discussion will move to a higher plane that revolves around interpretation or applicability of real results of a real case rather than speculation based on hypothetical situations.

4.5 The Next Step

Well, you have your input. Most likely you have a long list of things to do. Some people find the feedback discouraging; other people find it motivating. If you are convinced that your library is on the right track, be prepared to repeat the above procedure several times. The last draft of the Serialization library is # 20. About half of these revisions were posted to Boost and actually subjected to the process described above. The serialization library is larger than most and I really was not prepared for this process the way you will be after reading this paper. Hopefully, my experience qualifies as a worst case scenario.

5. FORMAL REVIEW

Formal review is the heart of the Boost process. It is fiendishly clever and very effective. It can be summarized as follows:

- Formal review is requested by submitter
- If the request is seconded by one or more Boosters, a limited time review period (usually a week or two) is scheduled and a review manager is assigned.
- Issues such as library design, utility, code quality, documentation, and others are discussed on the list.
- During the review period, interested parties post recommendations and supporting arguments for acceptance or rejection of the library into Boost.
- After close of the review period, the review manager makes the decision as to whether or not the library will be accepted, and if so, what changes should be made. His report includes a summary of the issues raised and his assessment of the consensus.

A particularly intriguing aspect of this process (to me) was the lack of pretense to any sort of democratic idea. Although reviews often "Vote" for acceptance or rejection, it is not a question of number of votes. The review manager makes the final decision after reviewing all the posted comments. It is much more akin to a court decision rather than an election.

The fact that this is a formal review will motivate a number of people who did not have time to review the library before to now take a closer look. Having updated your library in accordance with your preliminary feedback will pay big dividends here. Less time will be spent on mostly settled issues so you will be able to spend time on any new things that pop up.

The formal review process itself can be pretty intense for the library submitter. The limited time frame available focuses

everyone's attention on the review. Many new points will be brought up and you will have to consider them all in a short time. The process sounds more suspenseful than it really is. By the time this is done, it is usually obvious whether or not the submission will be accepted.

If your library is not accepted, the review manager's report will detail the reasons why along with the final decision "The X library is not accepted into Boost at this time". Of course such a decision is a huge disappointment for the submitter. Though it has happened that a library which was deemed unacceptable was reviewed a second time, it has happened only once. The Boost Serialization library is the holder of that dubious distinction. A better strategy is to be ready the first time by following the advice given here.

6. SO YOU THINK YOU'RE DONE?

If your library is accepted, it is usually subject to some conditions. Boost does not require the library to be totally complete to be accepted. Accepting only libraries that are ready for release would place an unreasonable burden upon potential contributors. Your next task is to make all changes that the review manager has determined are necessary. This can take quite a while.

Once all the changes are made, you can concentrate on other portability to other platforms. Boost emphasizes that support for older non-conforming compilers is not a requirement. Whether you choose to implement conformance workarounds may depend on the nature of your library. If your library is the next greatest template metaprogamming wizardry, it may not make sense to try to support older compilers. If it is a more prosaic application such as a TCP/IP stack, it might be more appropriate to support a wider range of compilers. It is up to you.

Buiding and testing with other compilers, libraries and platforms can be more difficult that one might think. First of all, if you have made it this far, your library may have lot more functionality and generality than it started with. You will start to gain a better appreciation for the subtleties of the C++ language and the variations among implementations of the language. Eventually you will add your code to the main Boost CVS tree and start testing on other platforms. Boost runs all the tests for all the libraries approximately every 24 hours. The slow turn-around can be infuriating. Fortunately, friendly Booster members interested in your library will often lend a hand with the compilers, libraries and platforms that they use.

Eventually, most of the boxes in the test/compiler test matrix show the tests passing. A few will not pass because one or more compilers or standard libraries cannot support a particular feature that your library requires. (e.g., wide character I/O). Some compiler bugs just cannot be worked around, so some feature of your library may not be usable with a particular compiler. This matrix will help library users to determine which features are available in their development environment.

Is this the end? Not really. Libraries are constantly tested and new problems emerge as compilers are upgraded. Users report ever more bugs or ambiguities in documentation. Users post suggestions for enhancements. Depending on the size of the library and how widely used it is, it can take a while before things really taper off.

7. IS IT REALLY WORTH IT?

Submitting a library to Boost and seeing it through can take a lot of time. It can be frustrating and stressful as well. And for all this, there is a real possibility that the effort will end in failure. The question has to be asked – is it worth the effort?

Regardless of whether or not your library is accepted, you will benefit from having gone through it.

You will find that there is a lot more to C++ than you thought there was. As a library writer you will likely become a lot more familiar with the details of templates, STL, streams, etc. than you do as an applications developer.

You will be exposed to better methodology. The Zen of Boost might be summarized as

- Design, code, tests, and documentation are developed in parallel rather than one after the other.
- Development is incremental and iterative. During the course of development, one always has a complete working package.
- Subject code, tests, and documentation to constant review and criticism of one's peers
- Factor out common code into libraries of orthogonal functionality.
- Test each library and each library feature independently.
- Document libraries separately.
- Composing programs from working, tested, documented components increases the chances of producing flexible, reliable programs in the shortest time.

Most organizations believe that they are using the best practices to produce software. Most of these organizations are wrong. Going through this process – even for a small library – will make it apparent what it takes to do good work and why more of it is not being done.

You will spend time interacting with smart, mostly agreeable people who really love what you – and they – do.

Is this "worth it"? You decide.

8. ACKNOWLEDGEMENTS

David Abrahams and other Boost members critiqued this paper.

9. REFERENCES

[1] www.boost.org

[2] http://www.boost.org/LICENSE_1_0.txt

[3] http://www.Boost.org/more/discussion_policy.htm

[4] http://www.boost.org/libs/libraries.htm

[5] http://www.boost.org/libs/libraries.htm#Correctness

xpressive: Dual-Mode DSEL Library Design

Eric Niebler
Boost Consulting
1608 E Republican St. 202
Seattle, WA 98112
eric@boost-consulting.com

Abstract

A Domain-Specific Embedded Language (DSEL) is a miniature language-within-a-language for solving problems in a particular domain. This paper presents techniques for increasing the power and flexibility of DSELs in C++ by unifying two complementary designs: early-bound (compile-time) and late-bound (runtime). Late-bound DSELs often have string-based interfaces, whereas expression templates are usually the tool of choice for early-bound DSELs. xpressive, a new regular expression library, fuses these two approaches. This fusion, providing both a runtime and a compile-time interface, has advantages over either approach alone. The presented unified design uses the same back end for both styles of binding, maximizing code reuse without sacrificing performance. This paper covers the design of xpressive and the advantages of its dual-mode approach.

1 Introduction

Domain-Specific Embedded Languages raise the level of abstraction, allowing programmers to express solutions in a way that naturally suits the domain in which they are working. Examples include Blitz++ [22] for scientific computing, and Boost.Spirit [8] for parser generation. These two libraries use a technique known as *expression templates* [23] to define an embedded language within C++. There are advantages to this approach. In particular, expression template-based DSELs are:

1. Type safe: the rules for legal statements in the embedded language are checked by the compiler.
2. Efficient: by delaying evaluation of complicated expressions until the full expression is available, expression templates make the job of an optimizing compiler easier.

In a different approach to DSELs, one writes statements in the domain-specific language as strings to be parsed and interpreted at runtime. This approach also has advantages. In particular, string-based DSELs are:

1. Unconstrained: they need not satisfy the rules for legal C++ expressions.
2. Dynamic: statements in the domain-specific language can be specified at runtime.

A library that provides both a string-based and an expression template-based interface has the potential to offer the benefits of both. The design of such a library presents significant implementation challenges:

1. How to structure the code to get the performance benefits of early binding while allowing the flexibility of late binding.
2. How to avoid duplication of implementation.

These issues and others are addressed by xpressive, a new regular expression template library. xpressive allows programmers to write regular expressions either as strings, expression templates, or a combination of both.

2 Advantages of a Dual-Mode Interface

The regular expression library recently accepted into Technical Report 1 [18] provides the following interface for constructing a regular expression object:

```
// match a date of the form 09/30/2005
regex date = "\\d\\d?/\\d\\d?/\\d\\d(?:\\d\\d)?";
```

Although more verbose, expression templates guard against syntax errors such as unbalanced parentheses by moving their detection to compile-time. For example, when written as an expression template using xpressive, the regular expression above would look like:

```
sregex date
   = _d >> !_d >> '/'         // match month
     _d >> !_d >> '/'         // match day
     _d >> _d >> !(_d >> _d); // match year
```

In this regex, the primitive `_d` serves the same purpose as `"\\d"` in TR1 regex; that is, it matches a digit character, and the unary logical not operator marks a sub-expression as optional.

Programmers can also create named regex objects and treat them as aliases, embedding them in other regular expressions, as in the following:

```
// A line in a log file is a date followed by
// a space, and everything up to the newline.
sregex log = date >> ' ' >> +~set['\n'];
```

This regex reuses the `date` regex defined above.

Another advantage is that expression templates can call other C++ code. Consider this regex, which only matches valid dates:

```
sregex date
   = (_d >> !_d)[if_is_month()] >> '/'
     (_d >> !_d)[if_is_day()]   >> '/'
     (_d >> _d >> !(_d >> _d))[if_is_year()];
```

This regex uses the programmer-defined predicates `if_is_month()`, `if_is_day()` and `if_is_year()` to enforce semantic constraints on the regular expression.

xpressive also accepts regular expressions as strings. By doing so, xpressive preserves the benefits of a late-bound interface; in particular, programmers can use the ECMAScript standard regex syntax [14], and programs can process arbitrary regular expressions at runtime.

A regex can be largely fixed at compile-time while part of its behavior can be customized at runtime by changing an embedded dynamic regex. Since matching a date is locale-dependent, the regular expression required to match a date might be written as a string and put in a resource file for easy localization, as in:

```
// A line in a log file is a date followed by
// a space, and everything up to the newline.
sregex date = sregex::compile(get_date_pattern());
sregex log = date >> ' ' >> +~set['\n'];
```

In this case, `get_date_pattern()` reads a localized string from a resource or initialization file. It is "compiled" into a regular expression that is then embedded in the log file regular expression.

3 Design and Implementation

The core of xpressive is modular, connecting its components at compile-time to use static dispatch, or at runtime to use dynamic dispatch. Statically-bound heterogeneous data structures stand in for their dynamically-bound counterparts, and iterative runtime algorithms have recursive variants that operate on the heterogeneous data structures.

xpressive avoids code duplication by isolating the core functionality in a Matcher concept and defining two Scaffolds: one for binding sequences of Matchers statically and the other dynamically. The decoupling of the Matcher and Scaffold concepts permits the Matchers to be neutral regarding the binding, whether dynamic or static [6]. This separation of concerns enables the core pattern matching functionality to be shared by the two Scaffolds.

3.1 Concepts

Matchers accept a match context (which, among other things, contains the iterators designating the sequence being searched) and a tail parser. The use of a tail parser to implement exhaustive backtracking recursive descent is described in [12]. The Matcher concept[1] looks like this:

```
template<class X, class Iterator, class Tail>
concept Matcher
{
  where BidirectionalIterator<Iterator>,
    Scaffold<Tail, Iterator>;

  bool X::match(context<Iterator> &,
    Tail const &) const;
};
```

Scaffolds control the policy by which Matchers are bound; therefore, they do not need to be passed a tail parser as a Matcher does.

[1] The syntax used in this paper for concept definitions conforms to the proposal to add concepts to C++[20].

Scaffolds generally compose a Matcher and a tail parser (which itself satisfies the Scaffold concept), and it passes the tail parser to the Matcher. The Scaffold concept is defined (in part) below:

```
template<class X, class Iterator>
concept Scaffold
{
  where BidirectionalIterator<Iterator>;

  bool X::match(context<Iterator> &) const;
  // ...
};
```

`any_matcher` is an example of a concrete type that satisfies the Matcher concept. It matches any one character, like the `'.'` metacharacter in Perl.

```
struct any_matcher {
  template<class Iterator, class Tail>
  bool match(context<Iterator> & ctx,
    Tail const & tail) const {
    if(ctx.current == ctx.end)
      return false;
    ++ctx.current;
    if(tail.match(ctx))
      return true;
    --ctx.current;
    return false;
  }
};
```

In this code, `ctx.current` is an iterator pointing to the current position in the sequence, and `ctx.end` is the end of the sequence. All concrete Matchers are implemented similarly; they evaluate their match condition, update the match context, invoke `tail.match(ctx)` and, if the tail parser fails, backs out changes to the match context. As we will see, the call to `tail.match(ctx)` can be dispatched either statically or dynamically.

3.2 Late-Binding with the Dynamic Scaffold

The dynamic Scaffold is built like a singly-linked list of Matchers, where each Matcher is encapsulated behind a runtime polymorphic interface. This is essentially a variation of the Interpreter design pattern [11]. The dynamic Scaffold is implemented in two parts, as below:

```
template<class Iterator>
struct matchable {
  virtual ~matchable() {}
  virtual bool match(context<Iterator> &)
    const = 0;
};

template<class Matcher, class Iterator>
struct dynamic_scaffold : matchable<Iterator> {
  Matcher head;
  matchable<Iterator> const * tail;

  bool match(context<Iterator> & ctx) const {
    return head.match(ctx, *tail);
  }
  // ...
};
```

The parameter `Matcher` to the `dynamic_scaffold` template is assumed to satisfy the Matcher concept. As such, it has a `match()` member function that accepts a `context<>` and a tail parser. In this case, the tail parser passed to the Matcher is a `matchable< Iterator >`, which satisfies the Scaffold concept. Looking back at the implementation of `any_matcher::match()`, we can see that when it is invoked from a `dynamic_scaffold`, the call to `tail.match(ctx)` will be dispatched dynamically.

3.3 Early-Binding with the Static Scaffold

The static Scaffold is also built like a singly-linked list, except that it is heterogeneous and statically-bound [5]. Its type is calculated at compile-time from the expression template. The static Scaffold is implemented as follows:

```
template<class Matcher, class Tail>
struct static_scaffold {
  Matcher head;
  Tail tail;

  template<class Iterator>
  bool match(context<Iterator> & ctx) const {
    return head.match(ctx, tail);
  }
  // ...
};
```

The only difference between the `static_scaffold` and the `dynamic_scaffold` is that the `static_scaffold` knows the exact type of the tail parser. Looking again at the implementation of `any_matcher::match()`, we can see that when it is invoked from a `static_scaffold`, the call to `tail.match(ctx)` will be dispatched statically.

3.4 Handling Branches and Loops

The picture painted so far is obviously simplistic. A singly-linked list of Matchers is only sufficient for handling regular expressions that do not have branches (alternation) and loops (quantification, such as the Kleene star [13]). Loops introduce cycles into our data structure. It turns out that dealing with cycles is one of the most challenging problems when moving from a runtime polymorphic data structure to a statically-bound, heterogeneous data structure. In the compile-time world, such cyclic data structures lead to cycles in the type system, which are forbidden. (Consider the thorny infinite regress problem of trying to declare a `std::pair<First, Second>` where `Second` is a pointer to the whole `std::pair` structure.) xpressive uses a general technique for breaking cycles in the type system while preserving the cyclic flow of control. The technique, described below, uses a form of type erasure [19] that does not incur the performance overhead of an indirection.

Consider the static regular expression `+(`*`expr`*`)`, which matches *`expr`* one or more times, where *`expr`* is some regular expression. A simple approach might be to terminate *`expr`* with a special loop-end Matcher which, when wrapped in a `static_scaffold`, *`expr`* can invoke as a tail parser. This loop-end Matcher would need to store a pointer to *`expr`* so it can jump back to the start of the loop. We might naively implement `loop_end_matcher` as follows:

```
// BUGBUG this doesn't work! Why?
template<class Loop>
struct loop_end_matcher {
  Loop const *loop; // ptr to loop top

  template<class Iterator, class Tail>
  bool match(context<Iterator> & ctx,
    Tail const & tail ) const {
    if(loop->match(ctx))
      return true;
    return tail.match(ctx);
  }
};
```

Unfortunately, this doesn't work. The problem becomes obvious once we try to write down the type of the Scaffold for the regex `+(.)`:

```
static_scaffold<
  any_matcher,
  static_scaffold<
    loop_end_matcher< /* What goes here? */ >
    // ...
```

`loop_end_matcher` needs to store a pointer to the top of the loop, but its type depends on the type of the `loop_end_matcher` – a cycle in the type system! This naive design cannot be made to work.

We notice that we can break the cycle if we move the parameterization from the loop-end Matcher's *type* to the Matcher's `match()` member function. Instead of storing the `loop` pointer as a data member, `loop_end_matcher:: match()` can accept the pointer as a parameter, where its type will be deduced. This neatly side-steps the infinite regress problem of having to declare a self-referential type. In the general case, we would need a stack of such back-pointers to handle nested repeats such as `+(+(`*`expr`*`))`. For dynamically-bound regexes, a `std::stack< matchable< Iterator > const *>` would serve, and for static regexes, the stack would be heterogeneous and statically bound. In addition, rather than adding an extra parameter to the Matcher's `match()` function, the stack of back-pointers could be bound together with the tail parser.

Although theoretically sound, this approach hardly meets our requirements for a zero-overhead solution. Certainly, maintaining a `std::stack<>` in the dynamic case will slow things down, and even in the static case, a heterogeneous stack of back-pointers will take up valuable real estate on the program stack. We need a hybrid approach.

It is only by separating the runtime data (the value of the pointer) from the compile-time data (the pointer's type) that we can solve this problem efficiently. The values of the back-pointers are stripped from the stack, which now becomes no more than a typelist [3]. The typelist is used to decorate the type of the tail parser, which gets passed to the Matchers in the usual way. The values of the pointers are stored in a type-erased form within the Matchers that need them; a `void*` is sufficient. Essentially, it is as if we broke the cycle by parameterizing `loop_end_matcher` defined above on `void` instead of on the real type of the loop. To call through a back-pointer, a Matcher casts its `void*` to the type at the head of the typelist passed in, completing the cycle just in time, and calls through it. The result is a general, zero- overhead mechanism to preserve the cyclic flow of control in a data structure that, from the type system's perspective, is acyclic.

3.5 Maintaining the Typelist

Expressing our solution in code is straightforward. Since we will will be maintaining a stack of types during matching, we must extend the Scaffold concept with stack operations. Our new Scaffold concept looks like this:

```
template<class X, class Iterator>
concept Scaffold {
  where BidirectionalIterator<Iterator>;

  bool X::match(context<Iterator> &) const;

  template<class Top>
  where { Scaffold<Top, Iterator> }
  bool X::push_match(context<Iterator> &) const;

  bool X::pop_match(context<Iterator> &,
    void const *) const;

  bool X::top_match(context<Iterator> &,
    void const *) const;

  bool X::skip_match(context<Iterator> &) const;
};
```

The semantics of the new member functions are defined below:

`push_match()`: Push `Top` onto the head of the typelist and invoke `match()` on `*this`.

`pop_match()`: Let `Top` be the type at the head of the typelist and `top` be the result of casting the `void const*` argument to `Top const *`. Remove `Top` from the head of the typelist, and call `match()` on `top`.

`top_match()`: Let `Top` be the type at the head of the typelist and `top` be the result of casting the `void const*` argument to `Top const *`. Call `match()` on `top`, leaving `Top` at the head of the typelist.

`skip_match()`: Discard the type at the head of the typelist and call `match()` on `*this`.

We show that all branching and looping can be implemented in terms of these four primitive operations with zero additional overhead in both the static and dynamic dispatch scenarios.

The implementation of these primitives in the dynamic dispatch scenario is simplicity itself. The `matchable<>` template we saw earlier can implement these functions *in situ*, as follows:

```
template<class Iterator>
struct matchable {
  virtual ~matchable() {}
  virtual bool match(context<Iterator> &)
    const = 0;

  template<typename Top>
  bool push_match(context<Iterator> & ctx)
    const {
    BOOST_MPL_ASSERT((
      tr1::is_same<Top, matchable<Iterator> >));
    return this->match(ctx);
  }
  bool pop_match(context<Iterator> &ctx,
    void const *top) const {
    return static_cast<matchable<Iterator>
      const *>(top)->match(ctx);
  }
  bool top_match(context<Iterator> & ctx,
    void const *top) const {
    return static_cast<matchable<Iterator>
      const *>(top)->match(ctx);
  }
  bool skip_match(context<Iterator> & ctx)
    const {
    return this->match(ctx);
  }
};
```

In the dynamic dispatch scenario, all the back-pointers will have exactly the same type: `matchable<Iterator> const *`. Therefore, a typelist is totally superfluous and is eliminated. We can verify at compile time that no type besides `matchable< Iterator >` is pushed on the stack using a static assertion in `push_match()`. (The flavor of static assertion used above is from the Boost MPL [2].)

Things are more complicated for the `static_scaffold`. Implementing the Scaffold concept in a statically-bound data structure requires a helper class, called `stacked_scaffold`. Conceptually, a `stacked_scaffold` is 2-tuple consisting of a tail parser and a type. The type represents the head of the typelist, and the tail parser can be either a `static_scaffold` or a `stacked_scaffold`.

Recall that a `static_scaffold` has a Matcher sub-object called `head` and a tail parser called `tail`. When `push_match<Top>()` is called on a `static_scaffold`, it binds `tail` and `Top` into a temporary `stacked_scaffold` object and passes it as the tail parser to `head.match()`. On compilers that implement the Empty Base Optimization (EBO), we can play a small trick with inheritance and `static_cast` to avoid even creating the temporary object, which saves valuable program stack space[2]. The code in Appendix A shows how `static_scaffold` and `stacked_scaffold` work together to satisfy the Scaffold concept with zero runtime overhead.

3.6 Turing Completeness

Loosely speaking, a programming language is Turing complete if it can do sequence, branch and repetition [25]. Therefore, if we show that the Scaffold concept is sufficient to implement these three operations, we have shown that it can be used to perform any calculation. It is trivial to show that the Scaffold supports sequencing; calling `match()` on a Scaffold causes execution to be passed in turn to the Scaffold's tail parser, and so on until the end of the sequence is reached. Branching and repetition are more interesting.

Consider the regular expression `(a|b)c`, which matches `a` or `b`, followed by `c`, where `a`, `b`, and `c` are themselves regular expressions. Figure 1 shows how we can represent this structure with the Matcher and Scaffold concepts. The arrows represent pointers to the polymorphic base `matchable< Iterator >` when bound dynamically, and aggregation when bound statically. The rectangles represent special Matchers that control the flow of execution. There is an `alternate_matcher` which points to or aggregates the Scaffolds representing the regular expressions `a` and `b`. Both `a` and `b` are

[2] We make `stacked_scaffold` inherit from the tail parser it wraps. Since `stacked_scaffold` is empty otherwise, it will be layout-compatible with its tail parser if the compiler does EBO, so a cast is sufficient.

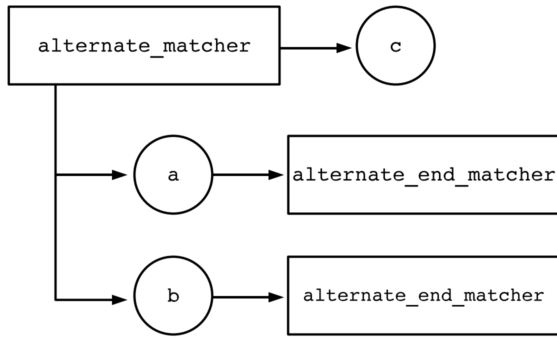


Figure 1. Data structure representing the regular expression `(a|b)c`.

terminated with an `alternate_end_matcher`. Flow control proceeds as follows:

1. `alternate_matcher` is invoked with `c` as a tail parser.
2. `alternate_matcher` calls `s_a.push_match<C>(...)` where `s_a` is the Scaffold representing the regular expression `a`, and `C` is the type of the Scaffold representing the regular expression `c`. This causes `a` to execute and pushes the type `C` to the head of the typelist.
3. If `a` succeeds, execution reaches its `alternate_end_matcher`, which stores a `void*` to `c`. `alternate_end_matcher` invokes `pop_match(...)` on the tail parser, passing the `void*`. This causes the `void*` to be cast to a `C*`, removes `C` from the front of the typelist, and executes `c`.
4. If `a` fails, or if the failure of `c` causes `a` to backtrack, `alternate_matcher` calls `s_b.push_match<C>(...)` where `s_b` is the Scaffold representing the regular expression `b`, and the process repeats.

Essentially, `alternate_matcher` is an *n*-way branch. The `push_match()` and `pop_match()` primitive operations give us a convenient way to express continuous control flow across a discontinuous data structure.

Looping is handled in a way similar to branching. We have already suggested the existence of a special loop-end Matcher. We will also need a loop-begin Matcher. When repeating a regular expression one or more times, the Scaffold representing it is book-ended with the loop-begin and loop-end Matchers. The loop-begin Matcher simply executes `tail.push_match< Tail (ctx)` to execute its tail parser and push its type onto the typelist. The loop-end Matcher will try to call `tail.top_match(ctx, pv)` to jump back to the start of the loop (where `pv` is a `void*` pointing to `tail`). If that returns false, it will return `tail.skip_match(ctx)` to break out of the loop and pass execution on to the rest of the regular expression. The code is below:

```
struct loop_matcher {
  template<class Iterator, class Tail>
  bool match(context<Iterator> & ctx,
    Tail const & tail) const {
      return tail.template push_match<Tail>(ctx);
  }
};

struct loop_end_matcher {
  void const * pv; // points to top of loop

  template<class Iterator, class Tail>
  bool match(context<Iterator> & ctx,
    Tail const & tail) const {
    if(tail.top_match(ctx, pv))
      return true;
    return tail.skip_match(ctx);
  }
};
```

We have not yet succeeded in building a Kleene star. The looping mechanism described above must execute the loop body at least once, and the Kleene star repeats an expression *zero* or more times. However, we can build a Kleene star out of the `loop_matcher` and the `alternate_matcher` already described. We take the regex to repeat, book-ended with begin- and end-loop Matchers. Then we use `alternate_matcher` to make it alternate with an `epsilon_matcher`, which is a null-transition. In other words, we transform `*(expr)` to `(+(expr) | epsilon)`. The epsilon branch gives us a way to skip over the loop body entirely if it fails to match at least once.

Since we can express sequencing, branching and repetition with the Scaffold and Matcher concepts, it follows that they can be used to express domain-specific embedded languages within C++ that are Turing complete, and which can be bound statically or dynamically in a way that incurs zero extra runtime overhead.

4 Empirical Results

We analyze the size and speed trade-offs of static and dynamic regular expressions. As static regular expressions have no virtual function calls or other indirections, we expect them to perform better than their interpreted dynamic brethren. This assumes a compiler smart enough to optimize the code generated by the expression template. It also fails to take cache and locality effects into account. As always, there is no substitute for an empirical test.

Performance Benchmark Method

Appendix 2 shows the comparative performance of static and dynamic xpressive. The test is broken into two scenarios: short matches and long searches. The regexes for the short matches are taken from The Regular Expression Library [1], a repository of practical, real-world regexes, so the hope is that the results are fairly representative. For the long searches, the time to find all matches in a long English test is measured. The text is the complete works of Mark Twain [21], and the patterns are taken from the performance suite of the Boost.Regex library [17]. To account for cache effects, each test is run ten times in succession, and the smallest time is reported for each.

Performance Benchmark Results

The results of the performance test are that for Visual C++ 7.1, static regexes are consistently faster than dynamic, by 13% on average. On GCC 4.0, results were mixed, with dynamic xpressive occasionally and inexplicably out-performing static xpressive. We

have no satisfactory explanation for these outlying data points, but we note that on the whole, static xpressive performs better.

We also analyze the effect of static regexes on executable size. We might expect executable size to drop when using expression templates because code is only generated for the DSEL features that are actually used. In contrast, a string-based DSEL, since it does not know at compile time which features are used, must generate code for all of them. In addition, when not using the string-based interface, the parser which turns a string into a regex is not needed. However, even though we only pay for the features that are used, with expression templates we must pay for them repeatedly. For example, using a look-ahead assertion in three different contexts will generate the look-ahead code three times with expression templates, but only once for a string-based DSEL. Also, the metaprogramming required to manipulate the expression template takes up space in the executable.

Executable Size Benchmark Method

Several different regular expressions are taken from The Regular Expression Library [1] and translated into static regexes. They are added one at a time to an otherwise empty source file. With each additional static regex, the file is compiled in release configuration, and the size of the resulting executable is noted. The same is done for dynamic regexes. The results are tabulated in Appendix 3. Correlating the number of static regexes to executable size is naive, since complicated regexes are likely to generate more executable code than simple ones. As a result, the table in Appendix 3 correlates expression template complexity versus executable size, where the expression template complexity is defined as the number of overloaded operators used in the expression template.

Executable Size Benchmark Results

We find that the executable size scales roughly linearly with the total expression template complexity. For dynamic regexes, the executable size is unsurprisingly independent of the number of regular expressions used. For programs with low expression template complexity, the executable size with static regexes is considerably smaller than with dynamic. For example, a program with only one static regex with a complexity of 12 results in a 57Kb executable, whereas the equivalent program using a dynamic regex is 156Kb. The break-even point is at a complexity of around 150. Beyond that, using dynamic regexes will yield smaller executables.

4.1 Recommendations

The above results can be summarized as follows: when optimizing for speed, prefer static regexes. If optimizing for size, take into consideration the complexity of the regular expressions. If the number of expression template operators is below a certain threshold (empirically determined to be around 150), use static regexes. As complexity grows beyond that threshold, consider switching to dynamic regexes. Once the "interpreter tax" has been paid, additional dynamic regexes are free. For applications that use a large number of regexes, a good strategy for managing executable size would be to use dynamic regexes for the majority, and use static regexes only where performance is critical.

5 Other Applications

Dual-mode DSEL interfaces have applications outside the domain of regular expressions. An interesting data point is the Spirit Parser Framework [8], which is an EBNF parser generator. An early version of the library exclusively used a string-based interface, but later versions switched to using an expression template interface. Joel de Guzman, Spirit's author, reports that users occasionally ask for an optional string-based interface, which de Guzman has considered adding.

Another domain that might benefit from a dual-mode DSEL is relational query. We imagine a library that accepts SQL queries as strings or as expression templates. When applied to a relational database, an expression template query could make it simpler to bind the results of queries to in-process data. The expression template queries could also be applied to strongly-typed in-memory data, such as STL container-like tables. In that case, the results of `SELECT` and `FROM` operations could themselves be strongly typed. In this regard, it would be like the Relational Template Library [24]. Such a dual-mode relational query library might also have the ability to translate an expression template query into an intermediate form (possibly string-based) for remote execution by a relational database.

Finally, we note the current work going on by Joel de Guzman on a new library called Rave [9], which in his words is "a lambda interpreter, which amounts to a late-bound DSEL for Phoenix." Phoenix [7] is an expression template-based lambda abstraction for C++, which is currently a part of Spirit.

6 Related Work

The idea of making the binding between components either dynamic or static is not new. Czarnecki and Eisenecker describe how to use parameterized inheritance to make a class fully statically-bound, fully dynamically-bound, or a combination of both [6]. xpressive uses a variation of this approach.

The idea of building DSLs out of pluggable components is also not new. Tools for building such mini-languages abound. Martin Fowler provides an excellent and informative overview of the state of the art in what he calls "language workbenches" in [10].

A variation of the cyclic type dependency problem is addressed by the Barton-Nackman trick [4]. A technique using template template parameters to break mutual dependencies between class templates in certain circumstances is described in [16]. However, neither technique fully addresses the cyclic type dependency issues of the sort that can arise in generative programming.

The Boost Lambda Library [15] is another example of an expression template-based DSEL that is Turing-complete. It provides a lambda abstraction for C++ with sequencing, branching and looping constructs, as well as variable assignment. This library does not have the same problem with cyclic type dependencies because its looping constructs are iterative instead of recursive as with xpressive. The recursion in xpressive is to satisfy the exhaustive backtracking requirement, which is implemented with recursive descent and tail parsers.

The Phoenix library [7] is another Turing-complete lambda abstraction for C++ which uses expression templates, but unlike the Boost Lambda Library, Phoenix allows recursion. Phoenix's solution for the cyclic type dependency problem is similar to xpressive's; however, it is not zero-overhead. Rather than storing type-erased pointers as data members, the pointers are passed as parameters so their types can be deduced. This consumes space on the program stack.

We speculate that the extra argument passing may also consume clock cycles and increase register pressure.

7 Conclusions

Domain-specific embedded languages are a powerful abstraction tool. By fusing the two common approaches to DSELs in C++, late-bound and early-bound, we can achieve the benefits of both without sacrificing performance or flexibility. We present implementation techniques for developing dual-mode DSEL libraries that maximize code reuse, and a general technique for breaking cycles in the type system for cyclic heterogeneous data structures. The concepts used by xpressive allow for Turing complete DSELs in C++ that allow either binding style with no extra runtime overhead.

8 Acknowledgements

We would like to thank Joel de Guzman and Rene Rivera, who proof-read an early draft of this paper.

We would also like to thank Douglas Gregor for reviewing the syntax of the concept definitions in this paper.

9 References

[1] The regular expression library. `http://www.regxlib.com`.

[2] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming*. Addison-Wesley, 2004.

[3] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[4] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1997.

[5] K. Czarnecki and U. Eisenecker. Metalisp. `http://www.prakinf.tu-ilmenau.de/~czarn/meta/metalisp.cpp`.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[7] J. de Guzman. Boost.phoenix. `http://spirit.sourceforge.net`.

[8] J. de Guzman. The spirit parser framework. `http://spirit.sourceforge.net`.

[9] J. de Guzman, 2005. private communication.

[10] M. Fowler. Language workbenches: The killer-app for domain specific languages. `http://www.martinfowler.com/articles/languageWorkbench.html`.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[12] D. Grune and C. J. Jacobs. *Parsing Techniques - A Practical Guide*. Ellis Horwood, 1990.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[14] E. International. *Standard ECMA-262: ECMAScript Language Specification*. ECMA International, 1999.

[15] J. Jarvi and G. Powell. The boost lambda library. `http://boost.org/doc/html/lambda.html`.

[16] L. Kettner. Comp 290-001: Algorithm library design: Lecture notes. `http://photon.poly.edu/~hbr/cs903-F00/lib_design/notes/advanced.html`.

[17] J. Maddock. Boost.regex. `http://boost.org/libs/regex/doc/index.html`.

[18] J. Maddock. A proposal to add regular expressions to the standard library. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1429.htm`.

[19] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[20] J. Siek, D. Gregor, R. Garcia, J. Willcock, J. JŁrvi, and A. Lumsdaine. Concepts for c++0x. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1758.pdf`.

[21] M. Twain. The entire project gutenberg works of mark twain. `http://www.gutenberg.org/etext/3200`.

[22] T. Veldhuizen. Blitz++. `http://www.oonumerics.org/blitz/`.

[23] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[24] A. Vertleyb and D. Arapov. Rtl: The relational template library. *C/C++ Users Journal*, March 2004.

[25] Wikipedia. Turing completeness — wikipedia, the free encyclopedia, 2005. [Online; accessed 29-September-2005].

A Implementation of the Static Scaffold

The implementation of `static_scaffold` is shown below. It shows how to achieve statically-bound cyclic control flow in a heterogeneous data structure with zero runtime overhead. This implementation assumes the empty-base optimization as an added optimization to conserve program stack.

Since `static_scaffold` represents the condition when no types have been pushed onto the typelist, it need not implement `pop_match()`, `top_match()`, or `skip_match()`. Calling these functions on a `static_scaffold` would be invalid in the same way that calling `pop()` on an empty `std::stack<>` is invalid. Also notice that `stacked_scaffold` does not need to implement `push_match()` since it will inherit a working version from `static_scaffold`.

```
///////////////////////////////////////////////
// class: stacked_scaffold
// purpose: a 2-tuple of a tail parser and a type
// requires: Top is Scaffold, Tail is Scaffold
// satisfies: Scaffold concept
//
template<class Top, class Tail>
struct stacked_scaffold : Tail {
  template<class Iterator>
  bool match(context<Iterator> & ctx) const {
    return Tail::template push_match<Top>(ctx);
  }
  template<class Iterator>
  bool top_match(context<Iterator> & ctx,
    void const * top) const {
    return static_cast<Top const *>(top)->
      template push_match<Top>(ctx);
  }
```

```
  template<class Iterator>
  bool pop_match(context<Iterator> & ctx,
    void const * top) const {
    return static_cast<Top const *>(top)->
      match(ctx);
  }
  template<class Iterator>
  bool skip_match(context<Iterator> & ctx) const{
    return Top::skip_impl(
      static_cast<Tail const &>(*this), ctx);
  }

  template<class Tail, class Iterator>
  static bool skip_impl(Tail const & tail,
    context<Iterator> & ctx) {
    return tail.template push_match<Top>(ctx);
  }
};

///////////////////////////////////////////////////
// function: decorate_scaffold
// purpose: decorates the type of a tail parser
//   with a type representing the top of the
//   scaffold stack
// requires: Tail is a Scaffold
// assumes: the empty-base optimization
//
template<class Top, class Tail>
inline stacked_scaffold<Top, Tail> const &
decorate_scaffold(Tail const & tail) {
  return static_cast<
    stacked_scaffold<Top, Tail> const &>(tail);
}

///////////////////////////////////////////////////
// class: static_scaffold
// purpose: binds a Matcher to a tail parser
// requires: Tail is a Scaffold
// satisfies: Scaffold concept (together with
//   stacked_scaffold)
//
template<class Matcher, class Tail>
struct static_scaffold {
  Matcher head;
  Tail tail;

  template<class Iterator>
  bool match(context<Iterator> & ctx) const {
    return head.match(ctx, tail);
  }
  template<class Top, class Iterator>
  bool push_match(context<Iterator> & ctx) const{
    return head.match(ctx,
      decorate_scaffold<Top>(tail));
  }

  template<class Tail, class Iterator>
  static bool skip_impl(Tail const & tail,
    context<Iterator> & ctx) {
    return tail.match(ctx);
  }
};
```

B Performance Benchmark

We analyze the relative performance of static and dynamic xpressive. The objective is to measure the cost of interpretation for dynamically bound regular expressions, and determine if the extra effort of authoring static regular expressions is worthwhile.

The tests were carried out on two different compiler/platform combinations: Microsoft Visual C++ 7.1 on Windows and GCC 4.0 on Linux. Two different scenarios are tested: (1) matching a short string against a regular expression, and (2) finding all matching substrings in a long Engligh text. The text is the complete works of Mark Twain [21], which is approximately 15Mb long. For all tests, the search is repeated in a loop until at least 0.5s has elapsed. The time to complete the search is taken to be the total time elapsed divided by the number of times the loop was executed. This process is repeated 10 times and the lowest number is reported. Each table of results has the actual time for both static and dynamic xpressive. It also has the normalized time, which is the actual time divided by whichever of the two times was lower. (Therefore, the best normalized time is 1.)

Table 1 shows the results of performing various short matches using the Visual C++ compiler. The regular expressions are from an online repository of useful regexes [1], so we have reason to believe they are fairly representative of how people actually use regular expressions. In this test, we can clearly see static xpressive consistently outperforming dynamic xpressive. It is important to note that for both static and dynamix xpressive, the code executing is the same, the only difference being whether the Matchers are bound statically or dynamically. Therefore, the performance difference is a measure of the virtual function call overhead, lost inlining opportunities and worse locality of reference.

Table 2 shows the results of performing repeated searches in a long English text. In several cases, dynamic xpressive is just as fast as static xpressive. For those regexes, xpressive has found an optimization that results in an algorithmic improvement. The strength of the optimization drowns out the comparativly small difference between static and dynamic dispatch. In the absense of clever optimizations, however, static xpressive is again faster than dynamic, by as much as 30%.

Tables 3 and 4 show the results of the same tests as 1 and 2, run this time on Linux after compiling with GCC 4.0. The results are quite erratic. Although static xpressive usually beats dynamic by a comfortable margin, Table 3 holds a few surprises. For some patterns, when matching against short strings, dynamic xpressive executes faster by as much as 30%. We have no satisfactory explanation for these surprising results. Possible explinations include a clever compiler optimization, opportunistic cache effects, or a bug in xpressive, the test harness or the compiler. Further investigation is required.

C Executable Size Benchmark

Table 5 compares executable size for programs using static and dynamic regular expressions. The first column is the number of static regexes in the program. The second column is the total expression template complexity of the program, where expression template complexity is the number of overloaded operators used. The third column is the size of the resulting executable in bytes. The fourth column is the size of the same program using dynamic regexes. The compiler used is Visual C++ 7.1. The executable is compiled in re-

lease configuration.

We can see from Table 5 that for programs that use regular expressions sparingly, using static regexes can greatly save space in the resulting executable. However, the executable size grows roughly linearly with the number and complexity of the static regexes. Eventually, at an expression template complexity of about 150, a threshold is passed and dynamic xpressive yields smaller executables.

Table 1. Performance of Short Matches on Visual C++ 7.1

<table>
<tr><th>Static xpressive</th><th>Dynamic xpressive</th><th>Text</th><th>Regular Expression</th></tr>
<tr><td>1 (3.2e-007s)</td><td>1.37 (4.4e-007s)</td><td>100- this is a line of ftp response which contains a message string</td><td>^([0-9]+)(| |$)(.*)$</td></tr>
<tr><td>1 (6.4e-007s)</td><td>1.12 (7.15e-007s)</td><td>1234-5678-1234-456</td><td>([[:digit:]]{4}[-]){3}[[:digit:]]{3,4}</td></tr>
<tr><td>1 (9.82e-007s)</td><td>1.3 (1.28e-006s)</td><td>john_maddock@compuserve.com</td><td>^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$</td></tr>
<tr><td>1 (8.94e-007s)</td><td>1.3 (1.16e-006s)</td><td>foo12@foo.edu</td><td>^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$</td></tr>
<tr><td>1 (9.09e-007s)</td><td>1.28 (1.16e-006s)</td><td>bob.smith@foo.tv</td><td>^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$</td></tr>
<tr><td>1 (3.06e-007s)</td><td>1.07 (3.28e-007s)</td><td>EH10 2QQ</td><td>^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$</td></tr>
<tr><td>1 (3.13e-007s)</td><td>1.09 (3.42e-007s)</td><td>G1 1AA</td><td>^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$</td></tr>
<tr><td>1 (3.2e-007s)</td><td>1.09 (3.5e-007s)</td><td>SW1 1ZZ</td><td>^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$</td></tr>
<tr><td>1 (2.68e-007s)</td><td>1.22 (3.28e-007s)</td><td>04/01/2001</td><td>^[[:digit:]]{1,2}/[[:digit:]]{1,2}/[[:digit:]]{4}$</td></tr>
<tr><td>1 (2.76e-007s)</td><td>1.16 (3.2e-007s)</td><td>12/12/2001</td><td>^[[:digit:]]{1,2}/[[:digit:]]{1,2}/[[:digit:]]{4}$</td></tr>
<tr><td>1 (2.98e-007s)</td><td>1.03 (3.06e-007s)</td><td>123</td><td>^[-+]?[[:digit:]]*\.?[[:digit:]]*$</td></tr>
<tr><td>1 (3.2e-007s)</td><td>1.12 (3.58e-007s)</td><td>3.14159</td><td>^[-+]?[[:digit:]]*\.?[[:digit:]]*$</td></tr>
<tr><td>1 (3.28e-007s)</td><td>1.11 (3.65e-007s)</td><td>-3.14159</td><td>^[-+]?[[:digit:]]*\.?[[:digit:]]*$</td></tr>
</table>

Table 2. Performance of Long Searches on Visual C++ 7.1

<table>
<tr><th>Static xpressive</th><th>Dynamic xpressive</th><th>Regular Expression</th></tr>
<tr><td>1 (0.019s)</td><td>1 (0.019s)</td><td>Twain</td></tr>
<tr><td>1 (0.0176s)</td><td>1 (0.0176s)</td><td>Huck[[:alpha:]]+</td></tr>
<tr><td>1 (1.78s)</td><td>1.1 (1.95s)</td><td>[[:alpha:]]+ing</td></tr>
<tr><td>1 (0.344s)</td><td>1.32 (0.453s)</td><td>^[^]*?Twain</td></tr>
<tr><td>1 (0.0576s)</td><td>1.05 (0.0606s)</td><td>Tom|Sawyer|Huckleberry|Finn</td></tr>
<tr><td>1 (0.164s)</td><td>1.16 (0.191s)</td><td>(Tom|Sawyer|Huckleberry|Finn).{0,30}river|river.{0,30}(Tom|Sawyer|Huckleberry|Finn)</td></tr>
</table>

Table 3. Performance of Short Matches on GCC 4.0

<table>
<tr><th>Static xpressive</th><th>Dynamic xpressive</th><th>Text</th><th>Regular Expression</th></tr>
<tr><td>1 (3.29e-07s)</td><td>1.35 (4.43e-07s)</td><td>100- this is a line of ftp response which contains a message string</td><td>^([0-9]+)(| |$)(.*)$</td></tr>
<tr><td>1.3 (6.96e-07s)</td><td>1 (5.34e-07s)</td><td>1234-5678-1234-456</td><td>([[:digit:]]{4}[-]){3}[[:digit:]]{3,4}</td></tr>
<tr><td>1 (8.11e-07s)</td><td>1.41 (1.14e-06s)</td><td>john_maddock@compuserve.com</td><td>^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$</td></tr>
<tr><td>1 (6.96e-07s)</td><td>1.56 (1.09e-06s)</td><td>foo12@foo.edu</td><td>^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$</td></tr>
<tr><td>1 (7.15e-07s)</td><td>1.47 (1.05e-06s)</td><td>bob.smith@foo.tv</td><td>^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$</td></tr>
<tr><td>1 (2.77e-07s)</td><td>1.14 (3.15e-07s)</td><td>EH10 2QQ</td><td>^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1} {0,1}[0-9][A-Za-z]{2}$</td></tr>
<tr><td>1 (2.77e-07s)</td><td>1.16 (3.19e-07s)</td><td>G1 1AA</td><td>^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1} {0,1}[0-9][A-Za-z]{2}$</td></tr>
<tr><td>1 (2.81e-07s)</td><td>1.12 (3.15e-07s)</td><td>SW1 1ZZ</td><td>^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1} {0,1}[0-9][A-Za-z]{2}$</td></tr>
<tr><td>1 (2.91e-07s)</td><td>1.08 (3.15e-07s)</td><td>04/01/2001</td><td>^[[:digit:]]{1,2}/[[:digit:]]{1,2}/[[:digit:]]{4}$</td></tr>
<tr><td>1 (3e-07s)</td><td>1.08 (3.24e-07s)</td><td>12/12/2001</td><td>^[[:digit:]]{1,2}/[[:digit:]]{1,2}/[[:digit:]]{4}$</td></tr>
<tr><td>1.18 (3.15e-07s)</td><td>1 (2.67e-07s)</td><td>123</td><td>^[-+]?[[:digit:]]*\.?[[:digit:]]*$</td></tr>
<tr><td>1.24 (3.43e-07s)</td><td>1 (2.77e-07s)</td><td>3.14159</td><td>^[-+]?[[:digit:]]*\.?[[:digit:]]*$</td></tr>
<tr><td>1.26 (3.43e-07s)</td><td>1 (2.72e-07s)</td><td>-3.14159</td><td>^[-+]?[[:digit:]]*\.?[[:digit:]]*$</td></tr>
</table>

Table 4. Performance of Long Searches on GCC 4.0

<table>
<tr><th>Static xpressive</th><th>Dynamic xpressive</th><th>Regular Expression</th></tr>
<tr><td>1 (0.0294s)</td><td>1 (0.0294s)</td><td>Twain</td></tr>
<tr><td>1 (0.0331s)</td><td>1 (0.0331s)</td><td>Huck[[:alpha:]]+</td></tr>
<tr><td>1 (1.16s)</td><td>1.1 (1.28s)</td><td>[[:alpha:]]+ing</td></tr>
<tr><td>1 (0.212s)</td><td>1.29 (0.275s)</td><td>^[^]*?Twain</td></tr>
<tr><td>1 (0.0519s)</td><td>1.12 (0.0581s)</td><td>Tom|Sawyer|Huckleberry|Finn</td></tr>
<tr><td>1 (0.13s)</td><td>1 (0.13s)</td><td>(Tom|Sawyer|Huckleberry|Finn).{0,30}river|river.{0,30}(Tom|Sawyer|Huckleberry|Finn)</td></tr>
</table>

Table 5. Executable Size

Count of Regexes	Expression Template Complexity	Size (Static)	Size (Dynamic)
1	12	57,344b	155,648b
2	24	65,536b	155,648b
3	68	81,920b	155,648b
4	87	94,208b	155,648b
5	100	102,400b	155,648b
6	115	110,592b	155,648b
7	116	110,592b	155,648b
8	120	122,880b	155,648b
9	123	122,880b	155,648b
10	129	126,976b	155,648b
11	133	139,264b	155,648b
12	149	159,744b	155,648b
13	202	172,032b	155,648b