

# Library Composition and Adaptation using C++ Concepts

Jaakko Järvi

Texas A&M University  
College Station, TX, U.S.A  
jarvi@cs.tamu.edu

Matthew A. Marcus

Adobe Systems, Inc.  
Seattle, WA, U.S.A  
mmarcus@adobe.com

Jacob N. Smith

Texas A&M University  
College Station, TX, U.S.A  
jnsmith@cs.tamu.edu

## Abstract

Large scale software is composed of libraries produced by different entities. Non-intrusive and efficient mechanisms for adapting data structures from one library to conform to APIs of another are essential for the success of large software projects. *Concepts* and *concept maps*, planned features for the next version of C++, have been designed to support adaptation, promising generic, non-intrusive, efficient, and identity preserving adapters. This paper analyses the use of concept maps for library composition and adaptation, comparing and contrasting concept maps to other common adaptation mechanisms. We report on two cases of data structure adaptation between different libraries, indicating best practices and idioms along the way. First, we adapt GUI controls from several frameworks for use with a generic layout engine, extending the application of concepts to run-time polymorphism. Second, we develop a transparent adaptation layer between an image processing library and a graph algorithm library, enabling the efficient application of graph algorithms to the image processing domain.

**Categories and Subject Descriptors** D.3.3 [Programming Languages]: Language Constructs and Features—Polymorphism; D.1.0 [Programming Techniques]: General

**General Terms** Design, Algorithms

**Keywords** generic programming, C++, software libraries, polymorphism

## 1. Introduction

Modern software systems commonly make use of components from a variety of software libraries. Software libraries at programmers' disposal are typically developed by different entities without centralized control. Consequently, interfaces of different libraries are seldom compatible as such. Using several libraries in one program can require significant amounts of "glue code" between components of different libraries. The amount and complexity of the necessary glue code affects the cost of using an existing software library for a particular problem. For complicated glue code the costs can become prohibitive—it may be easier to rewrite the needed components rather than to write the glue code to interface to them, or the composition mechanism may incur unacceptably high performance costs.

The language constructs and idioms for adaptation vary greatly between different programming languages, and can impact the cost of adaptation. This paper discusses library composition with the adaptation mechanism offered by C++ "concepts" [17], a set of extensions to C++ template system, likely to be included to the next revision of standard C++. Concepts augment C++'s template system with constrained templates. From here on, we refer to C++ extended with concepts as *ConceptC++*; C++ refers to the language as specified in its current standard [21].

Generic interfaces in *ConceptC++* are defined using the language construct **concept**. Component adaptation can be implemented using the **concept\_map** language construct. Concept maps are non-intrusive to the data structures they adapt. Consider a data structure defined in one library such that the data structure satisfies semantically, but not syntactically, the interface requirements of routines in another library. With concept maps it is possible to define a transparent adaptation layer between the libraries such that values from the first library can be directly used as input to the routines of the second: wrapping values explicitly into objects of another type is not necessary.

Adaptation with concept maps is efficient—combined with standard compiler optimizations, mainly inlining, adaptation with concept maps can be free of any performance cost. In contrast to interfaces defined using object-oriented abstract base-classes, generic interfaces defined using concepts do not directly support run-time polymorphism.

Currently *ConceptGCC* [16] is the only compiler for *ConceptC++*. Large parts of the C++ standard library have been implemented in *ConceptC++*, but otherwise use of the new features is modest. An efficient and flexible adaptation mechanism has been a design goal for *ConceptC++*, but sufficient evaluation of whether this goal has been met is lacking, as are patterns and idiomatic uses of the features for library composition. Furthermore, C++ standard library's collection of generic algorithms and data structures, formerly called the Standard Template Library (STL) [46], was the central use case that influenced the design of *ConceptC++*; the natural next step is to explore the applications of these new features to broader domains. Our goals for this work are thus to (1) evaluate how *ConceptC++* supports complex library composition by exercising its new features in a real-world setting, (2) to develop the idioms and patterns for effective adaptation between libraries, (3) to evaluate the performance implications of the new features, and (4) to compare and relate the features to other existing adaptation mechanisms in C++ and in other languages.

The structure of the paper is as follows. Section 2 briefly summarizes the paradigm of *generic programming*, the origin of concepts and concept maps. *ConceptC++*'s language constructs **concept** and **concept\_map**, and their use with constrained templates, is explained as well. Section 3 demonstrates the use of concept maps for adapting generic components in the domain of graphical user interfaces, and discusses idioms for combining run-time polymor-

© ACM, 2007. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in the Proceedings of the 6th International Conference on Generative Programming and Component Engineering (Salzburg, Austria, October 01 - 03, 2007). GPCE '07 <http://doi.acm.org/10.1145/1289971.1289984>

phism with the static polymorphism offered by concepts. Section 4 describes a complex library composition scenario where a transparent adaptation layer enables the use of an open-ended set of image types as input to a library of graph algorithms. We discuss the performance of such adaptation in Section 5. We relate concepts and concept maps to other adaptation mechanisms, such as instance declarations in Haskell and inheritance in object-oriented languages, in Section 6. Conclusions follow in Section 7.

## 2. Background

The design of ConceptC++ has mainly been motivated by the desire to better support the paradigm of *generic programming*, as practiced, e.g., in the design and implementation of the Standard Template Library, the Boost Graph Library (BGL) [41], the Matrix Template Library [42], the Adobe Source Library [3] and many other generic libraries in a variety of domains [4,9,13,40]. Generic programming is a systematic approach to designing and organizing software. It focuses on finding the most general (or abstract) formulations of algorithms together with their efficient implementations [23].

The generic programming approach to library design has proven to lead to efficient and reusable libraries. Characteristic to these libraries are rigorously specified interfaces, including both syntactic and semantic requirements. Roughly, syntactic requirements specify what operations must be supported by types to satisfy an interface, semantic requirements place constraints on the behavior and complexity of the operations. Generic programming strives for library interfaces that are complete in the sense that they capture the essential features necessary for implementing a class of efficient algorithms, and minimal in the sense that they are satisfied by many different data structures. For example, the STL and the BGL are both large libraries providing extensive functionality but the interfaces to these libraries are quite small. Through careful consideration of the essential requirements for certain classes of algorithms, the interface to a large number of library components has been made small and uniform. For these reasons the generic programming paradigm, and generic libraries, are of interest in the context of library composition—adapting a data structure for a particular library interface may open up a large part of the library for direct use. For example, the BGL, with a few dozens of lines of code, implements a transparent adaptation layer on top of graph data structures of the LEDA library [33], making the entire BGL usable for LEDA graphs without requiring any explicit wrapping or adaptation.

### 2.1 From C++ to ConceptC++

C++ templates are unconstrained. Generic C++ libraries therefore express the constraints on type parameters of generic algorithms as part of algorithms’ documentation. The STL established a systematic documentation style for this [5,45]. Sets of requirements on one or more types are referred to as *concepts*. (This is the reason for the naming of the language feature **concept** in ConceptC++.) Concepts describe the functions and operators that the types must support. They can also require a set of other accessible types, called *associated types*. Usually concepts also specify semantic requirements as algebraic laws that implementations of the functions and operators must satisfy, as well as upper bounds for the complexity of the functions and operators.

Despite the fact that concept descriptions are systematic, they are not understood by the C++ type checker. In C++, concepts are treated as mere documentation, though libraries providing some support for enforcing concept constraints have been proposed [32, 43]. ConceptC++ makes these descriptions known to the compiler, enabling modular type checking of templates and notably more informative and accurate compiler error diagnostics.

Type checking in ConceptC++ is not concerned with algebraic laws or complexity guarantees in concepts. Nevertheless, ConceptC++ does specify the syntax for expressing algebraic laws [17, §3.2.5] to serve as a hook for language tools.

### 2.2 Generic Programming in ConceptC++

This section briefly describes the new language constructs in ConceptC++. For a detailed description and specification, see [17, 19]. The central language construct of ConceptC++ is **concept**. Concepts define sets of requirements on a type, or on a tuple of types. We say that types that satisfy the requirements of a concept *model* that concept. For example, the following concept requires that the “less than” operator **<** is defined for objects of type **T**:

```
concept LessThanComparable<typename T> {
    bool operator<(T, T);
}
```

The operator **<** can be defined as a member function or as a non-member function defined in namespace scope; for some types it comes built-in. Concepts are not concerned with how the operator has been defined: any means is adequate to satisfy the requirement.

ConceptC++ requires an explicit declaration to establish that a particular type (or a parametrized class of types) is a *model* of a concept. The language construct used for these declarations are *concept maps*.<sup>1</sup> For example, the following two declarations state that the types **int** and **complex<double>** are models of the **LessThanComparable** concept:

```
concept_map LessThanComparable<int> {}
concept_map LessThanComparable<complex<double>> > {
    bool operator<(complex<double> a, complex<double> b) {
        return abs(a) < abs(b);
    }
}
```

The two definitions differ in how **LessThanComparable**’s requirements are satisfied. For **int**, the built-in **<** operator for integers satisfies the requirement for the **<** operator. For **complex<double>**, we explicitly provide a definition in the body of the concept map. For a concept map to type check, each required operation must either have a definition in the concept map’s body, or a definition must be found in the scope where the concept map is defined.

Concept maps can be templates. The following concept map declares all instances of the standard template **pair** to be models of **LessThanComparable**:

```
template <typename T, typename U>
requires LessThanComparable<T>, LessThanComparable<U>
concept_map LessThanComparable<pair<T, U>> {
    bool operator<(const pair<T, U>& a, const pair<T, U>& b) {
        return a.first < b.first || (!(b.first < a.first) && a.second < b.second);
    }
}
```

Explicit definitions of functions in the bodies of concept maps are a powerful tool for adaptation: consider the **complex<double>** type discussed above. The C++ standard library does not define **operator<** for complex number types. Therefore, **complex<double>** does not model **LessThanComparable**, which requires an implementation of **operator<** that defines a *strict weak ordering*. Complex numbers can, however, be defined a strict weak ordering, e.g. with the help of the **abs** function, as we do in the concept map above. In this adaptation, no change in the definition of the type **complex<double>** is necessary; neither is there a need to define a

<sup>1</sup> In earlier versions of the concepts extensions the possibly more descriptive keyword **model** was used. It was replaced with **concept\_map** which occurs far less frequently in existing C++ code.

```

template <typename Iter>
requires ForwardIterator<Iter>, LessThanComparable<Iter::value_type>
Iter min_element(Iter first, Iter last) {
    Iter best = first;
    while (first != last) {
        if (*first < *best) best = first;
        ++first;
    }
    return best;
}

```

**Figure 1.** The `min_element` generic algorithm.

wrapper type for `complex<double>` objects. The less than operator defined in the concept map is only visible in contexts constrained by the `LessThanComparable` concept.

Adaptation with concept maps is stateless. To be precise, certain features of C++, e.g. static local variables, enable the use of concept maps as stateful adapters—we do not discuss such uses.

Figure 1 shows an example of a simple generic algorithm `min_element` that uses the `LessThanComparable` concept as a constraint. Constraints on type parameters are stated in the `requires` clauses of templates. Bodies of templates are type checked assuming the constraints in the `requires` clauses hold. Correspondingly, at the time of template instantiation, the type checker checks that the template arguments satisfy the constraints in the `requires` clauses.

The `ForwardIterator` concept that appears in the constraints of `min_element` is shown in Figure 2. This concept provides basic iteration capabilities. The dereferencing operator `*` gives the value that an iterator refers to. The `++` operator advances an iterator to the next element. Equality comparison is used to decide when the end of the sequence is reached. Requirements for the operators `==` and `!=` are not stated directly in the body of `ForwardIterator`, but are obtained through *refinement* of another concept `EqualityComparable` (not shown). Syntax of refinement is that of inheritance between classes. The associated type `value_type` denotes the type of values that the iterator refers to. A `requires` clause in the body of a concept can place additional constraints on the parameters or associated types of a concept. Here, `value_type` must model `CopyConstructible`, which is one of the (draft) standard concepts and has its expected meaning. Examples of models of `ForwardIterator` include all pointer types and the iterator types of standard containers.

The `min_element` algorithm works for any sequence of values defined as a pair of iterators, as long as the iterator type is a model of the `ForwardIterator` concept and the iterator’s value type is a model of `LessThanComparable`. Assuming the concept map definition for `complex<double>` we showed above, the following call satisfies the constraints of `min_element`. The invocation of the less than operator in the body of `min_element` then calls the definition given in the concept map.

```

vector<complex<double> > cd;
// fill cd with values
complex<double> smallest = min_element(cd.begin(), cd.end());

```

A concept definition can be preceded with the keyword `auto`, signifying that no explicit concept map is necessary to establish a models relation between a type and a concept—structural conformance to the requirements suffices. Concept maps can, however, be written explicitly for `auto` concepts as well. Simple concepts with only a few requirements typically defined as `auto`. For example, the draft standard library uses `auto` in the definition of `LessThanComparable`:

```

auto concept LessThanComparable<typename T> { ... }

```

ConceptC++ provides a syntactic shortcut for succinctly expressing constraints. Type constraints can be stated directly in the

```

concept ForwardIterator<typename Iter> : EqualityComparable<Iter> {
    typename value_type;
    requires CopyConstructible<value_type>;

    value_type& operator*(Iter);
    Iter& operator++(Iter&);
    Iter operator++(Iter&, int);
}

```

**Figure 2.** The `ForwardIterator` concept (simplified from the one in the STL).

template parameter list: instead of the keyword `typename`, a concept name then precedes a template parameter. The shortcut is particularly convenient for single-parameter concepts. The following function signature uses the shortcut to constrain `Iter` to be a model of `ForwardIterator`:

```

template <ForwardIterator Iter>
requires LessThanComparable<Iter::value_type>
Iter min_element(Iter first, Iter last);

```

To the compiler, apart from parsing this function signature is no different from the signature in Figure 1.

A similar shortcut can be used with associated types. The declaration

```
CopyConstructible value_type;
```

can replace the following lines in Figure 2:

```

typename value_type;
requires CopyConstructible<value_type>;

```

We use both of these shortcuts in our examples.

### 3. Concept maps and adaptation

This section discusses the use of concept maps to adapt complex library interfaces encountered in a commercial software development environment.

An analysis carried out at a large commercial software development company revealed that roughly a third of the code and up to half of the reported bugs were related to the management of Graphical User Interfaces (GUI) [39]. There is little reuse across GUI related code between applications, and the features offered by the GUI frameworks tend to be used directly—higher level abstractions are rare.

One common task in the domain of GUIs is to layout widgets in a dialog box in a multi-lingual application. A single fixed layout is seldom suitable for many languages due, in part, to different glyph size and alignment characteristics. Manually maintaining multiple layouts, however, comes with a high cost. This makes a good case for the creation of a component to perform widget layout. There are many GUI frameworks with widely varying APIs for discovering the extents of widgets and for positioning them on screen. To support multiple platforms, and as a result of software evolution over time, it is not unusual for a suite of applications to support a half dozen GUI frameworks. A generic layout engine must be able to view and manipulate different widgets from different frameworks, making an adaptation layer necessary.

We discuss one such layout engine, Adobe’s *Eve*, a component of Adobe Source Library (ASL) [3]. ASL is Adobe’s generic open source library used in dozens of Adobe products. The layout engine calculates positions for a collection of widgets in a window, taking account of each widget’s size and alignment requirements. We refer to this information as the “extents” of a widget.

Figure 3 shows a simplified layout engine based on *Eve*. This simplified engine is written in terms of the operations defined by the `Placeable` concept:

```

template <Placeable P>
struct layout_engine {
    void append(P placeable) {
        placeables_m.push_back(placeable);
5    }

    void solve() {
        extents_t rect;
        extents_m.resize(placeables_m.size());

        for(int i = 0; i != placeables_m.size(); ++i)
10        measure(placeables_m[i], extents_m[i]);

        // "solve" layout constraints and update place_data_m

        for(int i = 0; i != placeables_m.size(); ++i)
            place(placeables_m[i], place_data_m[i]);
    }

15    vector<extents_t> extents_m;
    vector<P> placeables_m;
    vector<place_data_t> place_data_m;
};

```

**Figure 3.** A simplified layout engine modeled after Eve.

```

concept Placeable <typename T> : CopyConstructible<T> {
    void measure(T& t, extents_t& result);
    void place(T& t, const place_data_t& place_data);
}

```

Widgets are added to a layout problem using the **append** member function. The **solve** member function uses the **measure** operation from **Placeable** to query the extents of each widget (line 10), calculates a solution satisfying the layout constraints (not shown), and finally invokes the **Placeable**'s **place** operation to inform each widget of its calculated location (line 13). The three vectors **placeables\_m**, **extents\_m**, **place\_data** defined on lines 15–17 hold, respectively, the widgets to be placed, their extents, and the ultimately the positioning information for the widgets, as computed by the layout engine.

To measure a widget's size, in Apple's Carbon toolkit we might employ an API such as **GetBestControlRect**, whereas under Microsoft's Win32, one key step of this task is to invoke the **GetThemePartSizePtr** function. Code in Figure 4 adapts widgets from these toolkits to model the **Placeable** concept.

With these definitions in place, we are ready to exercise our layout engine. We illustrate in Figure 5 with driver code for the Carbon toolkit. First we parse the layout specification and create a collection of Carbon widgets storing alignment constraints in their user data area (line 5). Second, we instantiate the layout engine and populate it with the widget collection (line 6). When we ask the engine to **solve** (line 9) it must query each widget for its extents, solve the layout, and inform each widget of its final location. Finally, we make the window visible (line 10).

In sum, the concept map definitions we wrote adapt concrete GUI widget types to the API of our layout engine. As a result, we can directly instantiate the layout engine for any of the widget types for which we have written adapters.

### 3.1 Run-time polymorphism

The system presented above achieves transparent adaptation of widgets, but is too limited for our purposes. We want a layout engine that supports widgets from multiple toolkits simultaneously. Concepts and concept maps, however, do not directly support run-time variability. Most of the research and practice of generic programming in the context of C++ is concerned with the case where types of the inputs to generic algorithms are known at compile time—less emphasis has been placed on programming with concepts in cases where run-time variability is required. When programmers need run-time polymorphism they turn to inheritance

```

concept_map Placeable<HUIViewRef> {
    void measure(HUIViewRef p, extents_t& result) {
        Rect size;
        GetBestControlRect(p, &size, 0);
        // update result with size and other extents information
    }

    void place(HUIViewRef& p, const place_data_t& place_data) {
        Rect r;
        // place_data -> r
        SetControlBounds(p, &r);
    }
};

```

(a)

```

concept_map Placeable<HWND> {
    void measure(HWND p, extents_t& result) {
        SIZE r;
        // ...
        GetThemePartSizePtr(theme, dc, widget_type,
            kState, 0, measurement, &r);
        // update result with size and other extents information
    }

    void place(HWND& p, const place_data_t& place_data) {
        int X, Y, nWidth, nHeight;
        // place_data -> X, Y, nWidth, nHeight
        MoveWindow(X, Y, nWidth, nHeight, true);
    }
};

```

(b)

**Figure 4.** Adapters for Carbon (a) and Win32 (b) widgets.

and object oriented programming rather than concepts and generic programming. ASL, however, contains machinery that allows template functions constrained with concepts to be used when run-time polymorphism is required. This machinery is an extension of the ideas described in [38].

For our layout engine, in a multi-toolkit scenario, it will no longer be possible to instantiate the layout engine with a single fixed widget type. Instead, we can instantiate the engine with a widget wrapper type that augments concrete **Placeable** widgets with an external run-time dispatching facility. This adaptation layer is implemented as the class **poly<placeable\_rep>**, where **poly** template provides the concept independent boilerplate code for creating "run-time dispatching wrappers" and **placeable\_rep** the minimal concept dependent part. The **poly** template is described in details in [30]; for programmer documentation, see [3].

The **poly<placeable>** class nearly models the **Placeable** concept: it provides the **measure** and **place** operations, but it provides them as member functions, whereas **Placeable** requires the operations to be available as non-member functions. This disparity between mem-

```

layout_engine<HUIViewRef> le;
vector<HUIViewRef> mac_widgets;

HUIViewRef top_level_window =
5    parse_and_create_widgets("specification file", &widgets);
for(vector<HUIViewRef>::iterator i = mac_widgets.begin();
    i != mac_widgets.end(); ++i)
    le.append(*i);

le.solve();
10 ShowWindow(top_level_HUIView);

```

**Figure 5.** Driver code for Carbon toolkit.

```

auto concept PlaceableMF <typename T> : std::CopyConstructible<T> {
    void T::measure(extents_t& result);
    void T::place(const place_data_t& place_data);
};
template <PlaceableMF T>
concept_map Placeable<T> {
    void measure(T& t, extents_t& result) { t.measure(result); }
    void place(T& t, const place_data_t& place_data)
        { t.place(place_data); }
};

```

**Figure 6.** The member to non-member function adaptation idiom.

ber and non-member functions is a recurrent theme in C++. We can bridge this gap with an additional concept and a concept map. In our case, **poly-placeable** is adapted to conform to the **Placeable** concept with the code in Figure 6. **PlaceableMF** is a concept specifying that the **measure** and **place** operations are implemented as member functions. We adapt all models of the **PlaceableMF** concept to model the **Placeable** concept with the **Placeable<PlaceableMF>** concept map. In ASL we use the same idiom for thin wrappers, such as the *reference wrappers* in the (draft) C++ standard library [7, §20] and various *smart pointers* [2], to satisfy the requirements of a concept when their underlying type satisfies those requirements.

We can now replace the first line of the layout engine’s client code in Figure 5 with the instantiation:

```
layout_engine<poly<placeable_rep>> le;
```

There is no change in behavior; the window will be populated and laid out as before. We can now, however, use widgets from more than one toolkit at run-time.

In summary, the layout engine can be instantiated with any **Placeable** type. Our first use of concept maps is to adapt concrete widget types to become models of the **Placeable** concept. Our second use of concept maps is to adapt a run-time polymorphic widget wrapper—implicitly constructible from any **Placeable** concrete widget type—to be a model of **Placeable**, for use with the layout engine. These adaptations allow us to accommodate multiple GUI toolkits at run-time with no changes to the client code, the widgets, or the layout engine, all of which are possibly obtained as components of different software libraries.

The layout engine is generic, its parameters are constrained using concepts. It can be instantiated directly, or with an instance of the **poly** adapter to provide run-time polymorphism—clients of this library component decide whether to use run-time polymorphism or not. The adaptation layer providing run-time polymorphism is independent of other other adaptation layers; it can be “sandwiched” between multiple static adaptation layers that are implemented using concept maps.

## 4. Cross-domain composition

When a concept map adapts a particular type to model a concept, the concept map implements the concept’s operations in terms of the functionality provided by that type. In this section we move beyond adaptation of individual types to adaptation between entire library interfaces. In this situation, concept maps adapt collections of types. All types that model concept **A** are adapted to model concept **B** by a concept map that implements **B**’s required operations in terms of the operations provided by concept **A**. This kind of adaptation was already encountered in Figure 6, where calls to member functions are mapped to calls to non-member functions. We now explore a mapping between concepts that is much more complex than simply a direct translation between signatures: a mapping that adapts abstractions from one domain to those of another domain.

Our example is from the domains of image manipulation and graph algorithms. Many image algorithms can be viewed as graph algorithms given a suitable representation of images as graphs. In this section we present a partial composition of the Boost Graph Library (BGL) [41] and the Generic Image Library (GIL) [9]. We show the mapping from image related concepts defined in GIL to the graph concepts of BGL. Concept maps are instrumental for such cross-domain compositions. The adaptation code involves relatively few lines of code, is transparent to the client, and comes with minimal performance cost.

The Generic Image Library is Adobe’s open source image processing library, and also part of the *C++ Boost* [8] collection of peer-reviewed C++ libraries. The GIL defines concepts for raster images of any dimension, and provides generic implementations of basic image algorithms, such as copying, comparing, and applying a convolution. The GIL’s algorithms operate on an open-ended set of image types that may vary in color-space, pixel type, storage order, and other image characteristics.

The Boost Graph Library is a widely used library of generic algorithms for manipulating graphs. The BGL defines concepts that describe different capabilities for graph data structures, such as *incidence graphs* that provide access to the outgoing edges of each vertex, *vertex list graphs* that additionally allow access to all vertices in the graph, and *edge list graphs* that add the ability to access all edges in the graph. The BGL also provides useful data structures modeling these concepts, many implemented in terms of STL containers (essentially as compositions of vectors, lists, and maps). For BGL documentation see [44].

Neither BGL nor GIL are yet implemented using ConceptC++. We reimplement in ConceptC++ as much of the interfaces and implementations of these libraries as is necessary for our experiments. We omit support for mechanisms like BGL’s “named parameters”. BGL and GIL describe their algorithm requirements using STL-style concept documentation; our concepts are straightforward translations of this documentation into ConceptC++.

We focus on the *flood-fill* operation for images. Flood-fill transforms the color of a set of contiguous pixels that satisfy a predicate. The implementation of this algorithm performs a recursive search through neighboring pixels of an initial seed pixel. Applications of the flood-fill algorithm include transformation of a block of one color to another, insertion of a background texture (green screening), and image segmentation.

### 4.1 Implementation of the composition

The flood-fill algorithm is a breadth first graph search when an image is represented as a graph. In this representation, pixels correspond to vertices and each of the edges connect two vertices corresponding to neighboring pixels. BGL’s breadth first search algorithm imposes several concept requirements on its parameters. We can establish the image to graph correspondence directly with concept maps, by adapting concrete image types to model the BGL graph concepts. However, a more general adaptation for an open ended class of image types is achieved if we adapt the GIL image concepts to model BGL concepts. In both approaches the adaptation is not specific to flood-fill; many algorithms in the BGL use the same handful of concepts in their constraints.

The **breadth\_first\_search** function in our graph library is shown in Figure 7. For brevity, in all code examples we omit header includes, and the namespace prefixes of names from both GIL and BGL, as well as the prefix **std::** for names defined in the standard library. The **breadth\_first\_search** function is parametrized on the graph type, the type of queue used for storing references to vertices to maintain search state, a visitor type used for providing callback functions for various event points of the algorithm, and the type of color map used for tracking which vertices have already

```

template <typename G, typename Queue,
          typename Visitor, typename CMap>
requires IncidenceGraph<G>, Buffer<Queue>,
          BFSVisitor<Visitor>, ColorMap<CMap>,
          SameType<G, Visitor::graph>,
          SameType4<G::vertex_t, Visitor::vertex_t,
                  CMap::key_type, Queue::value_type>,
          SameType<G::edge_t, Visitor::edge_t>
void breadth_first_search (G const& g,
                          G::vertex_t const& s, Queue &Q, Visitor V, CMap Color);

```

**Figure 7.** The signature of the `breadth_first_search` function in the BGL. The somewhat verbose *same type* constraints guarantee that types of the function arguments are consistent, e.g., that the types of the values in the queue argument are the same as the types of the vertices in the graph.

been visited. The `breadth_first_search` function uses four concepts to constraint its template parameters. The `IncidenceGraph` concept specifies the requirements for the graph type: operations for enumerating out-edges of a given vertex, along with their incident vertices. The other concepts are `Buffer` that describes the operations of the vertex queue; `BFSVisitor` that specifies the dictionary of the callback functions; and `ColorMap` that defines the interface to the data structure storing vertex visitation information.

We focus on the `IncidenceGraph` concept, shown in Figure 8, in the description of the adaptation layer between images and graphs. The `Graph` concept, also in Figure 8, specifies associated vertex and edge types, which via refinement are provided by `IncidenceGraph` as well. Directly `IncidenceGraph` provides the `out_edges`, `out_degree`, `source`, and `target` operations, defined on lines 11–14. The `out_edges` function returns a pair of iterators that specify the sequence of edges emanating from a given vertex, and `out_degree` is for querying how many such edges there are. The associated types `out_edge_iterator` and `degree_size_type` on lines 6 and 7 have their expected meaning.

The `ImageView` concept in Figure 9 describes the interface that the GIL imposes on images. From the point of view of a type modeling a concept, operations specified in a concept are requirements that must be satisfied. From the point of view of an algorithm constrained by a concept, the operations are capabilities that can be relied upon. In our example, the GIL concepts’ capabilities are used to satisfy the BGL concepts’ requirements.

The `ImageView` concept on line 13 provides capabilities like those of STL containers: `value_type` is (usually) the type of the pixels, and the member functions `begin` and `end` return the iterator range for traversing the pixels of the image. The constraint on the `iterator` type is `LocatorIterator`, declared on line 8, that provides random access iteration and location services.

The `concept_map` that adapts `ImageView` to `IncidenceGraph` is shown in Figure 10. Lines 3–6 provide definitions for the associated types of `IncidenceGraph`. We represent vertices as pixel iterators. Edges are pairs, consisting of a pixel iterator and an integer. The first value of the pair is a GIL pixel iterator which specifies the source pixel, a point in an n-dimensional space. The second value of the pair encodes the direction that specifies which neighbor of the source pixel the target pixel is.

The `out_edges` function on lines 7–11 constructs the pair of edge iterators that denotes the sequence of out edges. The number of neighboring pixels, i.e. the number of out edges, for a given pixel is obtained as the `distance` between the beginning and end of the sequence of out edges. This formulation gives directly the implementation for the `out_degree` function on lines 17–20.

The first element of the pair representing an edge is the source vertex, and thus the implementation of `source` on line 12 is trivial.

```

concept Graph<typename G> {
    Regular vertex_t;
    Regular edge_t;
};
5 concept IncidenceGraph<typename G> : Graph<G> {
    InputIterator out_edge_iterator;
    UnsignedIntegral degree_size_type;
    requires Regular<out_edge_iterator>,
            SameType<out_edge_iterator::value_type, edge_t>,
10    SameType<out_edge_iterator::reference, edge_t&>;
    pair<out_edge_iterator, out_edge_iterator> out_edges (vertex_t, G);
    degree_size_type out_degree (vertex_t, G);
    vertex_t source (edge_t, G);
    vertex_t target (edge_t, G);
15 };

```

**Figure 8.** The `IncidenceGraph` concept. The `Regular` concept from the ASL specifies a type that is “well-behaved”, that is, it can be constructed, destructed, copied, assigned, and compared for equality. Furthermore, these operations respect obvious laws, such as a value and its copy comparing equal. The `InputIterator` and `UnsignedIntegral` concepts (not shown) are defined in the draft C++ standard library.

```

concept Locator<typename T> {
    SignedIntegral difference_type;
    difference_type axis (T);
    void axis (T&, difference_type);
5 long dimensions (T);
    bool valid (T);
};
concept LocatorIterator<typename T>
    : RandomAccessIterator<T>, Locator<T> {
10 requires SameType<RandomAccessIterator<T>::difference_type,
                Locator<T>::difference_type>;
};
concept ImageView<typename T> {
    typename value_type;
15 LocatorIterator iterator;
    requires SameType<iterator::value_type, value_type>;
    iterator T::begin () const;
    iterator T::end () const;
};

```

**Figure 9.** Capabilities provided by the GIL concepts. The concepts `RandomAccessIterator` and `SignedIntegral` (not shown) are defined in the draft C++ standard library.

The implementation of the `target` function (lines 13–16) is a bit more complex. GIL iterators have an *orientation*, which specifies the direction (horizontal or vertical in two-dimensional images) to move in an image when incrementing or decrementing the iterator. The GIL iterator that represents the source pixel of an edge is oriented along the axis towards the target pixel. The parity of the integer in the edge object, the second element of the pair, determines whether to decrement or increment the GIL iterator to find the target vertex.

To arrive at a flood-fill algorithm, we make one additional adaptation: we use a color map that is tailored to flood-fill, instead of BGL’s default color map for `breadth_first_search`. The color map stores the state of search by associating the status (as colors) of unseen (white), in progress (grey), or processed (black) to vertices. Only “white” vertices are added to the work queue. We associate the seed color with “white” and the desired output

```

template <ImageView Img>
concept_map IncidenceGraph<Img> {
    typedef Img::iterator vertex_t;
    typedef pair<vertex_t,int> edge_t;
5   typedef out_edge_adapter<Img::iterator> out_edge_iterator;
    typedef unsigned long degree_size_type;
    pair<out_edge_iterator, out_edge_iterator>
    out_edges (vertex_t const& v, Img) {
        out_edge_iterator oetr (v), oend (v, 2 * dimensions (v));
10    return make_pair (oetr, oend);
    }
    vertex_t source (edge_t const& e, Img) { return e.first; }
    vertex_t target (edge_t const& e, Img) {
        if (is_even (e.second)) return next(e.first);
15    else return previous(e.first);
    }
    degree_size_type out_degree (vertex_t const& v, Img) {
        pair<out_edge_iterator, out_edge_iterator> it = out_edges (v);
20    return distance(it.first, it.second);
    }
}

```

**Figure 10.** The `concept_map` adapting models of GIL `ImageView` to become models of BGL `IncidenceGraph`.

color with “black”. The default queue and visitor parameters of `breadth_first_search` that the BGL provides need no customization.

With the adaptations described above, the generic implementation of flood-fill in terms of the `breadth_first_search` becomes:

```

template <ImageView Img>
void flood_fill(const Img& img, Img::iterator start_pixel,
               Img::color target, Img::color replacement) {
    if (target == replacement || target != *start_pixel) return;
    deque<Img::iterator> queue;
    basic_bfs_visitor<Img> visitor;
    breadth_first_search(img, start_pixel, queue, visitor
                        image_colormap(target, replacement));
}

```

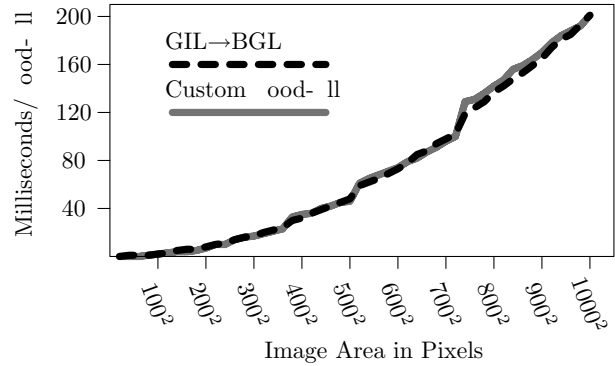
The queue and visitor parameters are those used in the BGL by default. We need to specify them explicitly since our graph library does not implement BGL’s named parameters mechanism.

## 5. Performance

Adaptation mechanisms can have a negative impact on performance. Mitchell et al. [34] give a detailed analysis of case where multiple inefficient adaptation layers had a major effect on performance of a large software system. In our examples we have used adaptation freely, adding layers as appropriate to meet our design goals. In this section we explore the performance costs of adaptation using concept maps.

We use the flood-fill algorithm as a test case and compare the execution times of two different test programs. The first program uses a flood-fill algorithm written directly in terms of the GIL concepts. Its implementation is essentially a breadth first search tailored for images. The second program uses concept maps to adapt GIL concepts to BGL concepts, and uses BGL’s `breadth_first_search` function, as described in Section 4. The direct implementation does not require an explicit representation of edges.

We compiled both test programs using the ConceptGCC [1] compiler, with the `-O3` flag on a MacMini Intel Core Duo, 1.67 GHz, with 2 GB of RAM. The test set consists of 50 square images, from the size of  $20 \times 20$  pixels to  $1000 \times 1000$  pixels. In each test program the flood-fill algorithm is used to fill the entire input image. Figure 5 shows the results. Both test programs have



**Figure 11.** The timing results.

essentially the same run-time performance. The implementation using cross-domain adaptation achieves performance on a par with a hand written GIL search algorithm.

The zero-overhead adaptation is due to C++’s template compilation model, where specialized code is generated for each different template instantiation. Calls to functions defined in concept maps can be statically resolved, and often inlined, allowing the optimizer to see through adaptation layers.

## 6. Concepts and other adaptation mechanisms

This section relates the adaptation capabilities offered by ConceptC++’s concepts and concept maps to those of several other mainstream languages, including C++.

Concept maps are a non-intrusive adaptation mechanism. To provide new functionality, or make a type conform to a new interface, no changes to the original definition of the type are needed. The adapters written as concept maps are stateless. Without trickery, such as static function scope variables, function definitions in the bodies of concept maps cannot store state and manifest different behavior in different call times.

The template system of C++, and ConceptC++, is based on instantiating templates with full type information at compile time, allowing all functions defined in concept maps to be statically resolved, possibly inlined, and further optimized. Several concept map adapters can be layered without the adaptation mechanism causing performance degradation. The downside is that all template instantiations to be used in a program must be known at compile time. While this may be acceptable in domains like graph algorithms or linear algebra—indeed, generic C++ template libraries have found widespread use in these domains— more “dynamic” domains, such as GUIs, necessitate run-time polymorphism.

In object-oriented languages, libraries publish their interfaces as abstract classes. Here, the term covers the “interface” language construct found in some languages. To satisfy the requirements of an interface then means defining a class that inherits from a particular abstract base class. This achieves run-time polymorphism but, in mainstream object-oriented languages, the subclass relation is established at the time of defining a class, which makes inheritance a relatively rigid mechanism for library composition. A class cannot retroactively, without altering its definition, be made to be a subclass of another class. Forms of structural subtyping have been proposed as cures for problems of rigid class hierarchies [6, 27] but have not found wide use. Amongst object-oriented languages, Cecil [28] lets one define subtype relationships outside of class definitions. Aspect-oriented programming systems can be used to modify classes retroactively, independently of their original definitions, e.g. to implement new interfaces using “static crosscutting” [25].

The *adapter pattern* [14] is widely used to work around problems of rigid class hierarchies when composing libraries. Adapters can be divided into object and class adapters. Both kinds of adapters inherit an abstract base class that defines the desired interface. Object adapters store the adaptee as a member (as a reference to a distinct object), whereas class adapters inherit from the adaptee, storing both the adapter and adaptee as a single object. The problems of library composition and adaptation in object-oriented programming are widely studied and recognized. For example, if there is a need to adapt a class with new functionality, but neither the definition of that class nor code that is hard-wired to use that class can be changed, an adapter is not an adequate solution (see, e.g., [31, 47]). Class adapters suffer from *hierarchy hardening* and object adapters from inconsistency problems caused by breaking the state of a single entity into multiple objects [20].

We demonstrate the techniques to combine run-time polymorphism with concepts in Section 3.1. Essential in our idioms is that we avoid the use of an abstract base class to describe the library interface. Instead, the library interface is specified in terms of concepts. As concept maps are entirely external to both the types they adapt from and the concepts they adapt to, the problems of object-oriented adapters are avoided. We use concept maps to adapt client code to and from the abstract base class interface, which is provided for the sole purpose of run-time polymorphism. The constructions to achieve this are somewhat involved, see [30], but can be hidden behind simple abstractions. The benefit is that the choice of whether to use run-time polymorphism is deferred to the time when the components are composed, rather than dictated by the library. Run-time dispatching may incur a performance penalty, which is thus avoided in the cases where run-time polymorphism is not needed.

Concepts are in many ways similar to Haskell type classes [49], and concept maps to Haskell’s instance declarations. A Haskell type class defines the signatures of the functions that instances (models) of the type class must implement. *Instance declarations* establish that a type, or a sequence of types in the case of multi-parameter type classes, belong to a particular type class. Analogous to concept maps, instance declarations are non-intrusive: external to both the definitions of the types and the definition of the type class. Lämmel and Ostermann collect formulations of problems reported in the object-oriented integration mechanisms [26], and demonstrate how type classes are effective solutions to many of them. Essential in evading the problems is the non-intrusive adaptation with instance declarations. Our experiences with non-intrusiveness of concept maps support this view.

In their standard form, type classes have a few obvious restrictions, which have largely been remedied in non-standard but common extensions. First, standard type classes only accept one parameter. Multi-parameter type classes, however, are widely supported by Haskell compilers and interpreters. Second, standard type classes do not support associated types. They can, however, be emulated to an extent with functional dependencies [24], a widely supported extension, or expressed directly using more recent extensions [11, 12].

There are also less obvious differences between concepts and type classes, some of which affect adaptation and library composition. We explain those differences, but refrain from a comparative evaluation; we have not produced Haskell implementations of any of the library composition and adaptation scenarios described in this paper. For comparative evaluation of the suitability of different mainstream languages to generic programming, see [15].

Haskell can infer the type class constraints of polymorphic functions automatically, while ConceptC++ does not support the analogous “concept inference”. To ensure that the constraints of a generic function can be uniquely determined, Haskell requires that an overloaded function name (when called without module qual-

```

data Canvas = ...
class Rectangle r where
  draw :: r -> Canvas -> Canvas
  move :: r -> Int -> Int -> r
class Cowboy c where
  draw::c -> c
  move::c -> Int -> Int -> c
  shoot::c -> c
drawShoot = shoot . draw

concept Rectangle<typename R> {
  void draw(R r, Canvas& w);
  void move(R& r, int x, int y);
}
concept Cowboy<typename C> {
  void draw(C& c);
  void move(C& c, int x, int y);
  void shoot(C& c);
}
template <Cowboy C>
void draw_shoot(C& c) {
  draw(c); shoot(c);
}

```

**Figure 12.** Accidental use of the same function name in two different type classes (left column) and in two difference concepts (right column).

ification) is declared in exactly one type class. When composing independently developed libraries, it is possible that the same function name is accidentally used in two type classes in different modules. Figure 12 translates the classic example of accidental conformance [29] to ConceptC++ and to Haskell. The Haskell version is erroneous and can be fixed by qualifying the calls to **draw** and **shoot** with the module prefix as **Cowboy.draw** and **Cowboy.shoot**; the ConceptC++ version is inevitably valid because ConceptC++ requires a disambiguating annotation, the “**Cowboy C**” constraint, regardless of whether conflicting concepts exist or not.

An instance declaration in Haskell is in effect in all functions in which the declaration is visible. A concept map, however, is only in effect in a context where a type is constrained with the corresponding concept. The example in Figure 13 illustrates. The operator **\*** for integers is defined differently in two different concept maps. The first concept map retains the **\*** operator’s original meaning, the second maps the operator to perform addition. Neither mapping has an effect outside generic functions. One or the other of the mappings, neither of them, or both can be in effect within a particular generic function, depending on the functions constraints. In our slightly contrived example function, both meanings apply.

Concept maps are a new layer on top of the existing overloading mechanism of C++: the application of concept maps is geared for adapting interfaces. Concept maps define views that are only active when requested, which is a desirable trait for adaptation and library composition. However, this may prove to be confusing as well, as it creates a rift between generic and non-generic functions.

The Scala programming language [37] provides external adaptation with rather different mechanics, *implicit parameters*, but with an outcome that is close to adaptation using type classes or concepts. An implicit parameter to a method can be left out in a call to the method. The Scala compiler attempts to find a unique best matching value for that parameter in the call’s context. When the implicit parameters represent dictionaries of functions, a fairly faithful emulation of type classes follows [36]. Furthermore, Scala *views* utilize implicit parameters to non-intrusively define implicit conversions between types. Views seem promising for implementing cross-domain compositions like we discussed in Section 4; a possible experiment for future work.

C++ (without concepts) allows the definition of efficient non-intrusive adaptation layers. For example, we mentioned BGL’s transparent adapters for LEDA graphs in Section 2. Breuer et al. [10] report on a cross-domain library composition between the domains of linear algebra and graph theory. They adapt several concepts from the Parallel Boost Graph Library [18] to concepts found in the Iterative Eigensolver Template Library [48]. Their implementation is in C++, and uses overloading and template spe-

```

concept Monoid <typename C, typename Tag> {
    C operator*(C, C);
    C identity();
}

class additive {}; class multiplicative {};

concept_map Monoid<int, multiplicative> {
    int identity() { return 1; }
}

concept_map Monoid<int, additive> {
    int operator*(int a, int b) { return a + b; }
    int identity() { return 0; }
}

template <InputIterator It, InputIterator It2, typename U>
requires SameType<It::value_type, It2::value_type>,
    Monoid<It::value_type, multiplicative>, Monoid<U, additive>,
    Assignable<U>, Convertible<It::value_type, U>
U inner_product (It i1, It i1e, It2 i2, U init) {
    for (; i1 != i1e; ++i1, ++i2)
        init = init * ((*i1) * (*i2));
    return init;
}

int main() {
    vector<int> v;
    v.push_back(3); v.push_back(5);
    cout << inner_product(v.begin(), v.end(), v.begin(), 100) << endl;
}

```

**Figure 13.** Concept maps are only in effect in contexts constrained by the corresponding concept. In the `inner_product` function, the multiplication between `*i1` and `*i2` comes from `Monoid<U, additive>`, and is therefore integer addition as defined by the concept map `Monoid<int, additive>`. The multiplication between `init` and the result of the “additive” element-wise multiplication comes from `Monoid<It::value_type, multiplicative>`, and is thus integer multiplications as defined by the concept map `Monoid<int, multiplicative>`. The `Assignable`, `Convertible`, and `InputIterator` concepts come from ConceptGCC’s implementation of the draft standard library. The `inner_product` function computes the inner product of two sequences, accumulating to an initial seed value `init`. When executed, the program outputs `134`.

cialization to achieve the necessary adaptation, not `concept` and `concept_map` constructs of ConceptC++. Non-intrusive adaptation in C++ relies in a host of tricky template techniques, such as *traits classes* [35] and conditional overloading using the *enable\_if* template [22]. Though C++ can support complex non-intrusive adaptation, the techniques are brittle.

## 7. Conclusions

In addition to improving modular type-checking and error diagnostics of template libraries, “concepts”, a forthcoming set of new features of C++, offers powerful mechanisms for adapting data types to specific library interfaces. This paper provides an analysis of this aspect of C++ concepts, and a description of their use in relatively complex cases of adaptation. We demonstrate that transparent adaptation of data structures to several library interfaces is possible and straightforward. Also, we show that adaptation can be applied between entire library interfaces, and that adaptation imposes no performance penalties, even in our complex adapters.

As future work, we plan to extend the composition of GIL and BGL to cover a larger set of concepts in these libraries, and thus a larger set of algorithms, and move our experiments to production versions of these libraries once full implementations in ConceptC++ become available.

The main benefits of ConceptC++’s adaptation mechanisms are non-intrusiveness (a type can be adapted to one or more interfaces without altering the definition of the type), flexibility (instead of single data types, a generic family of data types, or classes of data types described using concepts, can be adapted with a single adapter), and performance (adaptation is implemented using small functions whose addresses are statically resolved, and are thus inlinable and optimizeable).

The compilation model of C++ templates is that of generating specialized code for each instantiation of a template with different types. This does not change with the introduction of constrained templates to the language. The generic library interfaces defined in ConceptC++ do not directly support run-time polymorphism. We describe the idioms to combine run-time polymorphism and concepts. As a result, run-time polymorphism appears as simply another adaptation layer that can be used by those clients of the library that need it. Clients that do not need run-time polymorphism can instantiate library components directly.

We are working on several aspects of the combination of run-time polymorphism and static polymorphism of generic programming, such as modeling the notion of concept refinement at run-time, and minimizing the amount of concept-specific code in run-time polymorphic concept adapters. Early results of this work is reported in [30].

## 8. Acknowledgements

We are grateful for Doug Gregor and Sean Parent for many helpful discussions on the topic of the paper, and Doug for his work on ConceptGCC and inside insights to it. This work was supported in part by NSF grant CCF-0541014. We acknowledge the GAANN from the Department of Education fellowship for their support.

## References

- [1] (GCC) 4.3.0 20070330 (experimental) (Indiana University ConceptGCC alpha 7 prerelease).
- [2] *Boost Smart Pointers Library*. [www.boost.org/libs/smart\\_ptr](http://www.boost.org/libs/smart_ptr).
- [3] Adobe Systems, Inc. *Adobe Source Library*, 2005. [opensource.adobe.com](http://opensource.adobe.com).
- [4] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science*, pages 193–208. Springer, Aug. 2001.
- [5] M. H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
- [6] G. Baumgartner, M. Jansche, and K. Läufer. Half & Half: Multiple Dispatch and Retroactive Abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Ohio State University, 2002.
- [7] E. P. Becker. Working draft, standard for programming language C++. Technical Report N2009=06-0079, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Apr. 2006.
- [8] Boost. *Boost C++ Libraries*. <http://www.boost.org/>.
- [9] L. Bourdev and H. Jin. *Generic Image Library*, 2006. [opensource.adobe.com/gil](http://opensource.adobe.com/gil).
- [10] A. Breuer, P. Gottschling, D. Gregor, and A. Lumsdaine. Effecting parallel graph eigensolvers through library composition. In *Performance Optimization for High-Level Languages and Libraries (POHLL)*, Apr. 2006.
- [11] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *ICFP ’05: Proceedings of the International Conference*

- on *Functional Programming*, pages 241–253, New York, NY, USA, Sept. 2005. ACM Press.
- [12] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 2005. ACM Press.
- [13] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, a computational geometry algorithms library. *Software – Practice and Experience*, 30(11):1167–1202, 2000. Special Issue on Discrete Algorithm Engineering.
- [14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co., New York, NY, USA, 1995.
- [15] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17:145–205, Mar. 2007.
- [16] D. Gregor. ConceptGCC: Concept extensions for C++. <http://www.generic-programming.org/software/ConceptGCC>, 2005.
- [17] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '06)*, pages 291–310. ACM Press, Oct. 2006.
- [18] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 2005 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, pages 423–437, Oct. 2005.
- [19] D. Gregor and B. Stroustrup. Proposed wording for concepts. Technical Report N2193=07-0053, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Mar. 2007.
- [20] IBM Research. *Subject-oriented programming and the adapter pattern*. [www.research.ibm.com/sop/sopcadap.htm](http://www.research.ibm.com/sop/sopcadap.htm).
- [21] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages: C++*. Geneva, Switzerland, 2nd edition, Oct. 2003.
- [22] J. Järvi, J. Willcock, and A. Lumsdaine. Concept-controlled polymorphism. In F. Pfennig and Y. Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *LNCS*, pages 228–244. Springer Verlag, Sept. 2003.
- [23] M. Jazayeri, R. Loos, D. Musser, and A. Stepanov. Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloss Dagstuhl, Germany, Apr. 1998.
- [24] M. P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244, New York, NY, 2000. Springer-Verlag.
- [25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, New York, NY, June 2001.
- [26] R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 161–170, New York, NY, USA, 2006. ACM Press.
- [27] K. Läufer, G. Baumgartner, and V. F. Russo. Safe structural conformance for Java. *The Computer Journal*, 43(6):469–481, 2000.
- [28] V. Litvinov. Constraint-based polymorphism in Cecil: towards a practical and static type system. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 388–411, New York, NY, USA, 1998. ACM Press.
- [29] B. Magnusson. Code reuse considered harmful. *Journal of Object-Oriented Programming*, 4(3), Nov. 1991.
- [30] M. Marcus, J. Järvi, and S. Parent. Runtime polymorphic generic programming—mixing objects and concepts in ConceptC++. In K. Davis and J. Striegnitz, editors, *Multiparadigm Programming 2007: Proceedings of the MPOOL Workshop at ECOOP'07*, July 2007. To appear.
- [31] M. Mattsson, J. Bosch, and M. E. Fayad. Framework integration problems, causes, solutions. *Commun. ACM*, 42(10):80–87, 1999.
- [32] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, Oct. 2000.
- [33] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [34] N. Mitchell, G. Sevitsky, and H. Srinivasan. The diary of a datum: An approach to modeling runtime complexity in framework-based applications. In *Proceedings of the First International Workshop of Library-Centric Software Design (LCSD '05). An OOPSLA '05 workshop*, Oct. 2005. As technical report 06-12 of Rensselaer Polytechnic Institute, Computer Science Department.
- [35] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
- [36] M. Odersky. Poor man’s type classes. Presentation at the meeting of IFIP WG 2.8, Functional Programming, July 2006. <http://lamp.epfl.ch/~odersky/talks/wg2.8-boston06.pdf>.
- [37] M. Odersky. The Scala language specification: Version 2.0, draft march 17, 2006. <http://scala.epfl.ch/docu/files/ScalaReference.pdf>, 2006.
- [38] S. Parent. Beyond objects: Understanding the software we write. Presentation at C++ Connections, [opensource.adobe.com/wiki/index.php/Image:Regular\\_object\\_presentation.pdf](http://opensource.adobe.com/wiki/index.php/Image:Regular_object_presentation.pdf), Nov. 2005.
- [39] S. Parent. A possible future for software development. Keynote talk at the Workshop of Library-Centric Software Design 2006, at OOPSLA'06, Portland, Oregon, 2006. [lcsd.cs.tamu.edu/2006](http://lcsd.cs.tamu.edu/2006).
- [40] W. R. Pitt, M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss. The Bioinformatics Template Library—generic components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.
- [41] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [42] J. Siek and A. Lumsdaine. The Matrix Template Library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, Nov/Dec 1999.
- [43] J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, Oct. 2000.
- [44] J. Siek, A. Lumsdaine, and L.-Q. Lee. *Boost Graph Library*. Boost, 2001. [www.boost.org/libs/graph](http://www.boost.org/libs/graph).
- [45] Silicon Graphics, Inc. *SGI Implementation of the Standard Template Library*, 2004. <http://www.sgi.com/tech/stl/>.
- [46] A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, Apr. 1994. <http://www.hpl.hp.com/techreports>.
- [47] C. Szyperski. *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.
- [48] M. Troyer and P. Dayal. The Iterative Eigensolver Template Library. <http://www.comp-physics.org:16080/software/ietl/>.
- [49] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.