Science of Computer Programming 75 (2010) 596–614

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Programming with C++ concepts

Jaakko Järvi [a,*], Mat Marcus [b], Jacob N. Smith [a]

[a] *Texas A&M University, College Station, TX, USA*
[b] *Adobe Systems, Inc., Seattle, WA, USA*

ARTICLE INFO

Article history:
Received 2 February 2008
Received in revised form 9 December 2008
Accepted 2 January 2009
Available online 17 January 2009

Keywords:
C++
Concepts
Software libraries
Component adaptation
Generic programming
Polymorphism

ABSTRACT

This paper explores the definition, applications, and limitations of *concepts* and *concept maps* in C++, with a focus on library composition. We also compare and contrast concepts to adaptation mechanisms in other languages.

Efficient, non-intrusive adaptation mechanisms are essential when adapting data structures to a library's API. Development with reusable components is a widely practiced method of building software. Components vary in form, ranging from source code to non-modifiable binary libraries. The *Concepts* language features, slated to appear in the next version of C++, have been designed with such compositions in mind, promising an improved ability to create generic, non-intrusive, efficient, and identity-preserving adapters.

We report on two cases of data structure adaptation between different libraries, and illustrate best practices and idioms. First, we adapt GUI widgets from several libraries, with differing APIs, for use with a generic layout engine. We further develop this example to describe the *run-time concept* idiom, extending the applicability of concepts to domains where run-time polymorphism is required. Second, we compose an image processing library and a graph algorithm library, by making use of a transparent adaptation layer, enabling the efficient application of graph algorithms to the image processing domain. We use the adaptation layer to realize a few key algorithms, and report little or no performance degradation.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Modern software systems commonly make use of components from a variety of software libraries. Software libraries available to programmers are typically developed by different entities without centralized control. Consequently, different libraries' interfaces are seldom directly compatible. The cost and complexity of the code needed to combine libraries is significant, and can be prohibitively expensive. It may be easier to rewrite the needed components than to reuse them, or the performance overhead of the library composition mechanism may not be acceptable.

The language constructs and idioms for adaptation vary greatly between different programming languages, and can impact the cost of library composition. This paper discusses programming with C++ "concepts" [18], a set of extensions to the C++ template system likely to be included in the next revision of standard C++. We explore the applicability and limitations of these new features, particularly focusing on the use of concepts for library composition via non-intrusive component adaptation.

Concepts augment C++'s template system with constraints. In this paper, we will refer to C++ extended with concepts as *ConceptC++*; *C++ 2003* will be used to denote the language as specified in its current standard [28]. At the moment,

* Corresponding author.
E-mail addresses: jarvi@cs.tamu.edu (J. Järvi), mmarcus@emarcus.org (M. Marcus), jnsmith@cs.tamu.edu (J.N. Smith).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.01.001

ConceptGCC [23] is the only compiler for ConceptC++. Compiler and library vendors are meeting to finalize the feature, and additional implementations are expected to begin appearing in the near future.

The C++ standard library's collection of generic algorithms and data structures, formerly called the Standard Template Library (STL) [60], was the central use case that influenced the design of ConceptC++. Consequently, the first application of ConceptC++ was the STL. The natural next step is to explore the applications of the new language features to broader domains. In this paper we report on several new applications of concepts, taken from both commercial software and programming research, and on idioms that extend the use of concepts to support run-time polymorphism. Specifically, we (1) demonstrate how to adapt components in a non-intrusive and efficient manner using concepts, (2) show how to create systems that defer decisions about which components will be used polymorphically until application composition time, (3) guide programmers on how to effectively use the C++ concepts feature, (4) evaluate how ConceptC++ supports complex library composition by exercising its new features in an industrial setting, (5) evaluate the performance implications of the new features, (6) compare and relate the features to other adaptation mechanisms in C++ and in other languages, and (7) raise some issues and describe challenges faced when programming with concepts.

The structure of the paper is as follows. Section 2 briefly summarizes the paradigm of *generic programming* that has motivated the design of ConceptC++, and introduces the main features of ConceptC++. Section 3 begins by demonstrating how one of these features, the *concept map* language construct, can be used to adapt generic components. Then it illustrates how we can recover support for run-time polymorphism while remaining consistent with the non-intrusive, pay-as-you-go adaptation enjoyed when programming generically with concepts and concept maps. 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 inputs to a library of graph algorithms. Performance of such adaptations is discussed in Section 5. Section 6 relates concepts and concept maps to other adaptation mechanisms, such as instance declarations in Haskell and inheritance in object-oriented languages. Section 7 points out some limitations of concept maps. Conclusions follow in Section 8.

2. Background

The design of ConceptC++ has mainly been motivated by the desire to better support the paradigm of *generic programming*, as practiced, for example, in the design and implementation of the Standard Template Library, Boost Graph Library (BGL) [55], Matrix Template Library [57], Adobe Source Library [1] and many other generic libraries in a variety of domains [2,7,13, 52]. Generic programming is a systematic approach to designing and organizing software, that focuses on finding the most general (or abstract) formulations of algorithms together with their efficient implementations [32].

The generic programming approach to library design has been proven to support the production of efficient and reusable libraries. The interfaces of these libraries are specified abstractly in terms of their syntactic and semantic requirements, rather than in terms of concrete types and functions. Syntactic requirements specify which operations (and *associated types*, see Section 2.1) must be supported by types to satisfy an interface, while semantic requirements place constraints on the behavior and algorithmic complexity of the operations. Generic library interfaces 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, yet the interfaces to these libraries are quite small. Through careful consideration of the essential requirements for related classes of algorithms, the interface to a large number of library components has been made small and uniform. The generic programming paradigm, and generic libraries, are of interest in the context of library composition, since adapting a data structure to 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 some graph data structures of the LEDA library [43], making the entire BGL usable for LEDA graphs without requiring any explicit wrapping or adaptation [55, Section 14.3.5].

2.1. From C++ 2003 to ConceptC++

In C++ 2003, templates are unconstrained. Generic C++ 2003 libraries, therefore, generally express constraints on type parameters of generic algorithms as part of algorithms' documentation and as names of template parameters. The STL established a systematic documentation style for this [3,59], in which requirements on one or more types are collected into tables. These tables describe the functions and operators that the types must support. They can also require a set of other accessible types, called *associated types*. Naming conventions for template parameters are used to indicate the corresponding requirements table. These conventions, however, do not serve as tangible interfaces to express a contract between components—from the compiler's perspective, they only exist in the mind of the programmer.

Tangible interfaces between components are a key element in successful composition strategies. When interfaces are artifacts known to the compiler, a number of benefits result. First, it is possible for the compiler to check and enforce the contract expressed in the interface, to various extents, depending on the language. Next, semantic properties captured in interface specifications may enable code optimizations. Tangible interfaces also offer a place to bundle related signatures and constraints. This gives programmers the ability to create coherent artifacts about which readers can reason. Finally, tangible interfaces serve as places to collect documentation as part of programmer-friendly SDKs (software development kits), or for use by documentation generation tools. For example, the popular Doxygen source documentation tool offers

numerous features organized around classes, but the "defgroup" facilities[1] that one would use to document generic functions' requirements that are not tied to interface constructs of the host language, are relatively weak by comparison.

When introduced to generic programming with C++ 2003, programmers accustomed to object-oriented languages have difficulty when they fail to find component requirements in constructs analogous to the familiar abstract base classes. Such challenges are exacerbated by lengthy compiler diagnostics[2] that arise because the bulk of type checking of templates in C++ 2003 occurs late, at the time of their instantiation. ConceptC++ at last changes this for generic programming in C++. The **concept** language construct gives an *explicit* representation of the syntactic requirements tables, and it is precisely these concept definitions that serve as the tangible interfaces. Since concepts are analyzed by the compiler (in addition to the programmer), modular type checking of templates is possible; features such as concept-based overloading also become easier to use. For example, ConceptC++ can provide notably more informative compiler error diagnostics [18].

Note that the development of ConceptC++ was preceded by cleverly designed template libraries that emulated concepts. These libraries provide some support for expressing requirements tables programmatically, for enforcing constraints on template parameters expressed using those tables, and for rudimentary "type checking" of template bodies [42,58]. These techniques are brittle and expert-friendly, and have not found their way into wide use.

The STL's requirements tables also specify semantic requirements, as algebraic laws that implementations of required functions must satisfy, as well as upper bounds for the algorithmic complexity of these functions. ConceptC++ supports expressing algebraic laws [22, Section 14.9.1.4] in concepts, as *axioms*. Type checking in ConceptC++, however, is not concerned with concepts' axioms (except for insisting that the expressions in axioms themselves are well-formed). Compilers and programmers can use axioms to justify optimizations, and they serve as a hook for auxiliary language tools. We do not further explore the usefulness of axioms in this article.

2.2. Generic programming in ConceptC++

This section briefly describes the new language constructs in ConceptC++. More detailed description is available in the C++ concepts proposal overview [18], and in the current full specification of concepts [22]. The central new construct is **concept**. It defines a set 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 [20, Section 20.1.2] requires that the *less-than operator* (**<**), and the other comparison operators, are defined for objects of type **T**:

```
concept LessThanComparable<typename T> {
  bool operator<(const T& a, const T& b);
  bool operator>(const T& a, const T& b) { return b < a; }
  bool operator<=(const T& a, const T& b) { return !(b < a); }
  bool operator>=(const T& a, const T& b) { return !(a < b); }
}
```

ConceptC++ requires an explicit declaration, a *concept map*,[3] to establish that a particular type (or a parametrized class of types) models a concept. For example, the following definition states that the type **int** is a model of the **LessThanComparable** concept:

```
concept_map LessThanComparable<int> { }
```

Another concept map makes a user defined type **name** a model of **LessThanComparable**:

```
struct name { char* first; char* last; };

int namecmp(const name& n1, const name& n2) {
  int c = strcmp(n1.last, n2.last);
  if (c==0) return strcmp(n1.first, n2.first);
  else return c;
};

concept_map LessThanComparable<name> {
  bool operator<(const name& a, const name& b) {
    return namecmp(a, b) < 0;
  }
}
```

The two concept maps differ in how they satisfy the **LessThanComparable** concept's requirements. For **int**, the body of the concept map is empty; the built-in comparison operators for integers satisfy the four requirements of the **LessThanComparable**

1 www.stack.nl/~dimitri/doxygen/grouping.html.

2 We have experienced a 20 MB error message from a single client error in the use of an, admittedly very complex, template library.

3 The possibly more descriptive keyword "**model**" that was used in preliminary designs of ConceptC++, was replaced with the keyword **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;
}
```

Fig. 1. The **min_element** generic algorithm.

```
  concept ForwardIterator<typename Iter> : Regular<Iter> {
2   typename value_type;
    requires CopyConstructible<value_type>;
4   value_type& operator*(const Iter&);
    Iter& operator++(Iter&);
6   Iter operator++(Iter&, int);
  }
```

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

concept. For **name**, one of the required operations, the less-than operator, is defined in the body of the concept map. This suffices to make **name** a model of the **LessThanComparable** concept, since the concept's body provides *default implementations*, defined in terms of the less-than operator, for the other three required operations. A type can satisfy a requirement, for example the requirement for a less-than-operator, in any one of the following ways, in decreasing order of precedence: (1) the concept map can define the less-than operator explicitly, (2) the less-than operator can be defined as a member or non-member function (or in some cases as a built-in operation), (3) or a default implementation in the concept itself can provide a definition. In a well-formed concept map, each required operation is defined in at least one of these ways.

Explicit definitions of functions in the bodies of concept maps are a powerful tool for adaptation. For example, here, objects of the **name** class can be compared using the **namecmp** function, whose interface is similar to that of the **strcmp** function. The concept map above adapts **name** to satisfy the requirements of the **LessThanComparable** concept, and defines the less-than operator in terms of the existing **namecmp** function. In this adaptation, the definition of the type **name** does not need to be modified, and the objects of type **name** do not need to be wrapped by other types, to be comparable with the less-than operator.

Definitions in concept maps do not introduce functions or type names into the global scope. For example, the less-than operator defined in the above concept map is only visible in generic definitions constrained by the **LessThanComparable** concept.

Concept maps can be made generic with templates. For example, the following concept map declares all instances of the standard template **pair** to be models of **LessThanComparable**—as long as the element types of the pair are **LessThanComparable**. The constraints on the element types, the template parameters **T** and **U**, are stated in the *requires clause*, introduced with the **requires** keyword:

```
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);
  }
}
```

Fig. 1 shows a simple generic algorithm, **min_element**, that uses the **LessThanComparable** concept as a constraint. Constraints in the requires clause are assumed to hold during type checking of the template's body, and they are enforced at the time of template instantiation. The **ForwardIterator** concept that appears in the constraints of **min_element** is shown in Fig. 2. This concept provides basic iteration capabilities. The indirection operator (∗) gives the value that an iterator refers to. The increment operator (**++**) advances an iterator to the next element. Equality comparison is used to decide when the end of a sequence is reached. Finally, iterators can be copied and assigned. Requirements for the equality operator (**==**) and the inequality operator (**!=**), as well as for copying and assignment, are not stated directly in the body of **ForwardIterator**, but are obtained through *refinement* of another concept **Regular**. A concept, **D**, is said to refine another concept, **B**, when all of **D**'s requirements are included in **B**'s requirements. The syntax for expressing refinement relationships resembles that used for expressing class inheritance relationships. The **Regular** concept (not shown, see [20, Section 20.1.7]) collects several common requirements supported by most types, including equality comparison, and the abilities to copy and assign objects. 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

```
vector<name> names;
// fill names with values
name first_in_line = min_element(names.begin(), names.end());
```

Fig. 3. A call to the generic **min_element** function.

model **CopyConstructible**, a self-explanatory concept from the standard library draft of ConceptC++. 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 delimited by a pair of iterators, as long as the iterator type is a model of the **ForwardIterator** concept and the iterator's associated **value_type** is a model of the **LessThanComparable** concept. In the next few paragraphs, we illustrate how type checking works for the call to **min_element**, shown in Fig. 3.

The C++ standard library specifies that **vector**'s **begin** and **end** member functions shall return types that satisfy the **ForwardIterator** concept. (In fact, these types satisfy stronger constraints, but this is not important for our purposes.) This satisfies the first requirement on types supplied to **min_element**. The second requirement, that the **iterator**'s associated **value_type** is a model of the **LessThanComparable** concept, is satisfied via the concept map for **LessThanComparable<name>** given earlier. As both these requirements are, collectively, met by the **vector<name>**'s iterator type and the **name** type, the call to **min_element** in Fig. 3 passes type checking.

To illustrate the role of concept maps in type checking, and as adapters, consider the call to the less-than operator, "**∗first < ∗best**", in the body of **min_element**. The type checker resolves this call to the **LessThanComparable** concept's less-than operator, looked up in the concept map **LessThanComparable<Iter::value_type>**. In the example of Fig. 3, at template instantiation time the placeholder associated type **Iter::value_type** is bound to the **name** type, and thus the call to the less-than operator is resolved to **LessThanComparable<name>::operator<(∗first, ∗best)**, which is implemented in terms of **namecmp** in the **LessThanComparable<name>** concept map.

Indirections through concept maps are efficient. Recall that in C++ 2003's template compilation model distinct code is generated whenever a template is instantiated with different types—this compilation model is used in ConceptC++ as well, after type checking a template instantiation. Once the type checker has accepted that the types bound to the template arguments satisfy the template's constraints, code specialized for the particular template instance, **min_element<vector<name>::iterator>** is generated. Consequently, the calls that depend on template parameters, such as the less-than operator call above, are statically resolved and subject to inlining and other compiler optimizations. This applies, in particular, to calls directed through functions in concept maps. Gregor and Siek give a more detailed account of type checking and compiling ConceptC++'s constrained templates [21].

A concept definition can be preceded with the keyword **auto**, signifying that no explicit concept map is necessary to establish an is-a-model-of relation between a type and a concept—it suffices that all functions and operations required by the concept are defined for the type. Concept maps can, however, also be written explicitly for **auto** concepts. Simple concepts, with only a few requirements, are typically defined as **auto**; we could define **LessThanComparable** as:

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

Throughout this article, we use ConceptC++'s syntactic shortcuts for succinct expression of constraints: instead of the keyword **typename**, a concept name can precede a template parameter in a template parameter list, or an associated type in the body of a concept. To demonstrate the former use of the shortcut, the signature of **min_element** in Fig. 1 can be re-written as:

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

As an example of the latter use, lines 2 and 3 in the body of the **ForwardIterator** concept in Fig. 2 can be replaced with the declaration "**CopyConstructible value_type;**".

3. Non-intrusive composition of components

To succeed in building software from components, some of which may be unmodifiable, a non-intrusive composition mechanism is necessary. In this section we take advantage of features of ConceptC++ (concepts and concept maps) that support such compositions. We then go beyond what is directly supported by ConceptC++ to develop library techniques that encapsulate run-time polymorphism as an additional non-intrusive adaptation layer. We illustrate our techniques using library interfaces abstracted from a commercial software development code base.

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) [49]. There is little reuse across GUI-related code between applications, and the procedures exposed in GUI library APIs tend to be used directly—higher-level abstractions are rare. One common task in the domain of GUIs is to place widgets in a dialog box in a multi-lingual application. A single fixed layout is seldom suitable for multiple languages due, in part, to different glyph size and alignment characteristics. The manual maintenance of multiple layouts incurs a high cost. This motivates the need for

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

Fig. 4. The **Placeable** concept.

```
   template <Placeable P> struct layout_engine {
 2   void append(P placeable) { placeables_m.push_back(placeable); }
     void solve() {
 4     extents_m.resize(placeables_m.size());
       for(int i = 0; i != placeables_m.size(); ++i) measure(placeables_m[i], extents_m[i]);
 6     // 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]);
 8   }
     vector<extents_t> extents_m;
10   vector<P> placeables_m;
     vector<place_data_t> place_data_m;
12 };
```

Fig. 5. A simplified layout engine modeled after Eve.

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

Fig. 6. Adapters for Carbon (a) and Win32 (b) widgets.

an automated layout engine component. We discuss one such layout engine based on Adobe's *Eve*, a component of Adobe Source Libraries (ASL) [1]. ASL is Adobe's generic open source library used in dozens of Adobe products.

The layout engine's task is to calculate positions for a collection of widgets in a window, taking into account each widget's size and alignment requirements. We refer to this information as the "extents" of a widget. GUI libraries have widely varying interfaces for discovering the widgets' extents and for positioning the widgets 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 depend upon a half-dozen GUI libraries. A generic layout engine must, therefore, be able to operate on different widget types from different libraries: a unified interface to interact with widgets and an adaptation layer to adapt different widget types to this interface are necessary. We represent this unified interface with the **Placeable** concept, shown in Fig. 4, whose **measure** and **place** functions capture the ability to query a widget's extents and to inform the widget of where it should ultimately place itself in the layout.

Fig. 5 outlines our generic layout engine, a simplified version of the Eve engine. The engine is parametrized on its widget type, which must satisfy the requirements imposed by the **Placeable** concept. The **append** member function adds widgets to a layout "problem." The **solve** member function uses the **measure** operation from **Placeable** to query the extents of each widget (line 5), then it 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 7). The three vectors **placeables_m**, **extents_m**, and **place_data_m**, defined on lines 9–11 hold, respectively, the widgets to be placed, their extents, and ultimately the positioning information for the widgets, as computed by the layout engine.

To use the layout engine with a particular widget type, the widget-specific behavior must be adapted to the interface defined by the **Placeable** concept. For example, when using Apple's Carbon API one might employ a routine such as **GetBestControlRect** to measure a widget's size, whereas when using a Microsoft Win32 themed widget set, a key step in this task is to invoke the **GetThemePartSizePtr** function. Code in Fig. 6 shows the concept maps that adapt widgets from each of these libraries to satisfy the requirements of **Placeable.** Fig. 7 illustrates the role of these concept maps in a diagram.

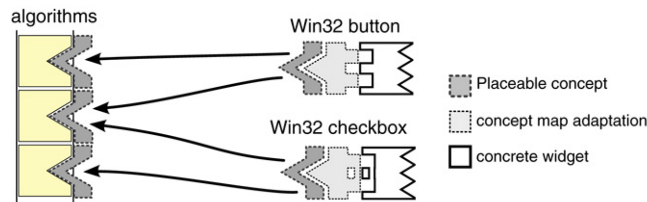

Fig. 7. Concept maps can non-intrusively adapt a given widget type to a particular generic library interface. The widget type is bound to the interface at compile time.

```
layout_engine<HIViewRef> le;
vector<HIViewRef> mac_widgets;
HIViewRef top_level_window = parse_and_create_widgets("specification file", &widgets);
for(vector<HIViewRef>::iterator i = mac_widgets.begin(); i != mac_widgets.end(); ++i) le.append(*i);
le.solve();
ShowWindow(top_level_HIView);
```

Fig. 8. Layout engine client code (Carbon API).

With these definitions in place, we are ready to exercise our layout engine. Fig. 8 displays typical layout engine client code when using the Carbon API. First the layout specification is parsed and a collection of Carbon widgets, with alignment constraints in their user data area (line 3), is created. Second, the layout engine is populated with the widget collection (line 4). When asked to `solve` (line 5), the engine queries each widget for its extents, solves the layout, and informs each widget of its final location. Finally, the window is made visible (line 6).

In sum, the concept maps in Fig. 6 for the `Placeable` concept each adapt a particular GUI widget type to the API of the generic layout engine. As a result, the layout engine can be directly instantiated and used with any single widget type for which the appropriate concept map has been written.

3.1. From compile-time to run-time polymorphism

The above adaptation of widget types is transparent and non-intrusive but also limited since the widget type of the layout engine is determined at compile time. In practice, a single instantiation of a layout engine sometimes needs to support multiple widget types. When we want to support more than one widget type at run time, concepts, concept maps, and templates are not sufficient—they 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 fixed 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 often turn to inheritance and object-oriented programming rather than concepts and generic programming. Conversion of existing libraries to support desired run-time polymorphic use cases by the addition of abstract base classes with appropriate virtual functions involves significant changes to the library API and semantics; new base classes may be introduced and classes which may have been written to be used with value semantics now must be used with reference or pointer semantics to avoid slicing. This can lead to the introduction of smart-pointers (see, e.g. [11]) for object lifetime management, which as a side-effect can lead to unintended shared value semantics.

Redesigning the library API can, however, be avoided. We describe the *run-time concept* idiom, an idiom that employs type erasure to allow a single instantiation of a generic component to operate upon multiple types at run time—*neither the generic library component's interface, nor its implementation or semantics are altered*. This idiom involves the insertion of an abstract interface, an associated concept map, and some auxiliary artifacts between a concrete model of a concept and a generic component implemented against that concept. The presented machinery is an extension of work by Parent [48].

We will apply the run-time concept idiom to our layout engine. We describe the implementation of the run-time concept machinery, in particular the `poly` class template, below. Before we introduce `poly`, we draw attention to the run-time concept idiom's principal benefit: the transformation from compile-time to run-time polymorphism requires no modifications to the engine's API, nor to the engine's principal client code. For example, if the first line in the code in Fig. 8 is changed to:

```
layout_engine<poly<placeable>> le;
```

the rest of the code works as before. Similarly, the code implementing the layout engine remains untouched. Now, widgets from several libraries with different APIs can be added to a single instance of the layout engine, to be solved and placed. The end result is that the decision of whether a generic component should use compile-time or run-time polymorphism is made by the client of the generic component, not mandated by the component's implementation.

These benefits do not come without a cost—auxiliary code is required for each concept that is to support run-time polymorphism. In the remainder of this section, we describe the steps and code needed to create a minimal adaptation, from

the ground up, for the specific case of the **Placeable** concept. Section 3.2 explains how the **poly** template and its accompanying library reduce the amount of boilerplate, and describes some additional functionality that the library provides.

The adaptation of the **Placeable** concept requires three auxiliary classes and a concept map. The first class is an abstract base class that represents the **Placeable** concept's required functions as pure virtual functions; this **placeable_runtime_concept** class can be defined as follows:

```
struct placeable_runtime_concept {
  virtual void measure(extents_t& result) const = 0;
  virtual void place(const place_data_t& place_data) = 0;
  virtual ~placeable_runtime_concept() { }
};
```

The second class is an implementation class that inherits from **placeable_runtime_concept**. This class stores the concrete widget in a member variable and implements the **placeable_runtime_concept**'s virtual member functions **measure** and **place** by delegating to the corresponding operations upon the concrete widget type's concept. A separate implementation class is required for each different widget type, but since different widget types provide a uniform set of operations via their **Placeable** concept maps, we can implement the classes as instances of a single class template **placeable_model_adapter**:

```
template <Placeable P>
struct placeable_model_adapter : placeable_runtime_concept {
  placeable_model_adapter(const P& p) : concrete_placeable(p) { }

  virtual void measure(extents_t& result) const { return Placeable<P>::measure(concrete_placeable, result); }
  virtual void place(const place_data_t& place_data) {
    return Placeable<P>::place(concrete_placeable, place_data);
  }

  P concrete_placeable;
};
```

For example, this template is instantiated as **placeable_model_adapter<HIViewRef>** when adapting Carbon widgets; the member variable **concrete_placeable** has type **HIViewRef**, and the member functions delegate to the operations defined in the concept map **Placeable<HIViewRef>**, shown in Fig. 6(a).

With the two components described above, a pointer of type **placeable_runtime_concept∗** can dynamically refer to different widget types. A raw pointer, however, behaves in a different manner from a **Placeable** widget type required by the layout engine. For example, the layout engine can copy objects that it stores, and it expects that when copied they follow "deep copy semantics", instead of creating aliases to the same objects. The task of the third class in the adaptation layer is to encapsulate the raw pointer type in a wrapper that provides the deep copy semantics: the lifetime of the widget-specific adapter matches that of the wrapper, and the widget's copy constructor and assignment operators copy and assign the objects pointed to, not just the pointers. The definition of **poly_placeable** is as follows:

```
struct poly_placeable {
  template <Placeable P>
  poly_placeable(const P& p) : placeable_adapter_m(new placeable_model_adapter<P>(p)) { }
  poly_placeable(const poly_placeable& x); // deep copy
  poly_placeable& operator=(const poly_placeable& x);
  ~poly_placeable() { delete placeable_adapter_m; }
  void measure(extents_t& result) const { return placeable_adapter_m−>measure(result); }
  void place(const place_data_t& place_data) { return placeable_adapter_m−>place(place_data); }
private:
  placeable_runtime_concept∗ placeable_adapter_m;
};
```

We omit the definitions of copy and assignment; their implementations rely on existence of **clone** member functions in the **placeable_runtime_concept** and **placeable_model_adapter** classes, which for simplicity are not shown either. This is boilerplate that we can implement in a library, as described in Section 3.2.

The **poly_placeable** class serves another purpose. Its templated constructor makes any type that models **Placeable** implicitly convertible to **poly_placeable**, and hides the instantiation of the **placeable_model_adapter** template and the wrapping of the widget object with that adapter. As the result, when an object of type **poly_placeable** is expected, any **Placeable** widget type will do. For example, the **append** member function of **layout_engine<poly_placeable>** accepts objects of type **HIViewRef** or **HWND** as parameters. Thus, the client of the layout engine only has to know about the **poly_placeable** type; the rest of the machinery stays hidden.

One more task remains: **poly_placeable** itself is not yet a model of **Placeable**, which is required for the instantiation **layout_engine<poly_placeable>** to succeed. The definition **concept_map Placeable<poly_placeable> { ... }** establishes this. However, the **poly_placeable** class almost satisfies the requirements of **Placeable** already: it provides the **measure** and **place** operations, but

```
auto concept PlaceableMF <typename T> : 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); }
};
```

Fig. 9. The member to non-member function adaptation idiom.

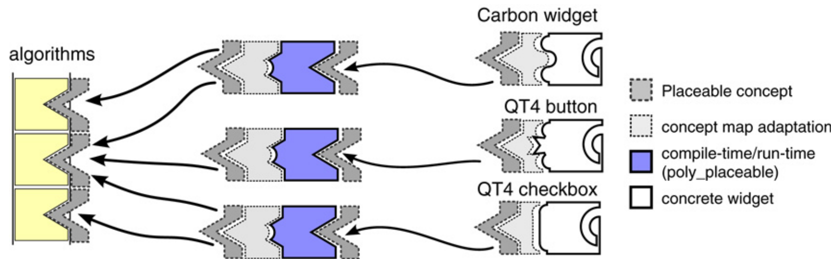

Fig. 10. Adaptation layers to achieve run-time polymorphism with templates: the layout engine operates on a type that models **Placeable**; concept maps adapt concrete widgets to satisfy these requirements; the run-time polymorphic **poly_placeable** wrapper can be constructed with any **Placeable** type; and a concept map again adapts the wrapper itself to be a model of **Placeable**.

it provides them as member functions, whereas **Placeable** requires the operations to be available as non-member functions. This disparity between member and non-member functions is a recurrent theme in C++. We can bridge this gap with an additional concept and a more general concept map template. Fig. 9 shows the **auto** concept **PlaceableMF** that specifies the requirements that **measure** and **place** operations are implemented as member functions. The **Placeable<PlaceableMF>** concept map adapts all models of the **PlaceableMF** concept to satisfy the requirements of the **Placeable** concept. This completes the remaining task, and makes **poly_placeable** a model of **Placeable**. In ASL we use the same idiom to allow thin wrappers, such as the *reference wrappers* in the (draft) C++ standard library [5, Section 20] and various *smart pointers* [11], to satisfy the requirements of a concept when their underlying type satisfies those requirements.

In summary, the machinery presented above provides a mechanism by which a programmer, acting as component assembler, can build systems from algorithms and classes defined in one or more generic libraries. Without requiring any modifications to these libraries, the component assembler enjoys the right to make decisions about which classes should behave in a run-time polymorphic manner. This use case is supported by the run-time concept idiom. The adaptation layer providing run-time polymorphism is independent of other adaptation layers; it can be "sandwiched" between multiple static adaptation layers that are implemented using concept maps. The adaptations implemented using concept maps do not impact object identity and minimize syntactic intrusion. In our example domain, as a result of the described idioms, we can accommodate multiple GUI libraries at run time with no changes to the client code, the widgets, or the layout engine, all of which may have originated as components of different software libraries. Fig. 10 illustrates the different layers of adaptation diagrammatically.

3.2. *The poly library*

The implementation of the adaptation layer presented in Section 3.1 is a bit laborious and detail-oriented. The concept being adapted, however, largely determines the implementation. The **Placeable** concept, for example, determines the definitions of the **placeable_runtime_concept**, **placeable_model_adapter**, and **poly_placeable**, that constitute a large amount of boilerplate. In practice the amount of boilerplate is even greater than what we sketched in our example—full support for run-time concept refinement, **dynamic_cast**-like querying and coercion, move semantics [24], deep copy, and equality comparison adds significant overhead. Besides the additional labor, the boilerplate is non-trivial code and thus a potential source of errors.

Concepts cannot be reflected upon in ConceptC++, so it is impossible to entirely avoid the boilerplate. We have, however, encapsulated a portion of the boilerplate in our **poly** library. The implementation of an adapter like **poly_placeable** with the **poly** library resembles the "hand-coded" implementation shown in Section 3.1—one still replicates the concept's requirements in an abstract class, provides a "model adapter" template, as well as a class similar to the **poly_placeable** handle class. These classes, however, need to declare and define only the methods corresponding to the requirements of the particular concept they represent; in the case of **Placeable**, these are **measure** and **place**. Moreover, the resulting adapter has additional functionality and performance enhancements that the **poly** library provides, as outlined below.

The **poly** library applies *small object optimization*: a small template meta-program that decides whether to store the concrete adaptee type in the body of the **poly<placeable>** adapter or allocate space for it on the heap. Support is also included

for moving temporary objects instead of copying them, when such support exists for the underlying concrete type. Move semantics, in the context of C++, refers to moving an expensive object from one address in memory to another with minimal expense, by "stealing" resources of the source object and leaving it in a destructible-only state [25]. C++'s type system provides means to detect many situations where a seemingly unsafe move operation can safely replace a (much more expensive) copy operation. The forthcoming revision of the C++ standard library will support move semantics for a large number of classes, such as the container classes.

In the `poly` library, inheritance of interfaces representing concepts is supported. In particular, poly objects conforming to a refined concept interface can be used transparently where poly objects conforming to a base interface are expected. The library also contains `poly_cast` operations that provide functionality for run-time concept refinement hierarchies that is analogous to the querying and coercion facilities provided by `dynamic_cast` in object-oriented hierarchies.

Marcus et al. describe the `poly` template in details [40]; the programmer documentation [1] describes a version of the library for C++ 2003. There are ways to optimize uses of the `poly` template so that virtual function dispatching is not always necessary [51].

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 Fig. 9, where calls to member functions are mapped to calls to non-member functions. We now explore a more complex mapping between concepts, one 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 [12,53,54]. In this section we present a partial composition of the Boost Graph Library (BGL) [55] and the Generic Image Library (GIL) [7] that enables many image processing algorithms to be implemented as simple wrapper functions over BGL algorithms. We show the mapping from image-related concepts defined in the GIL to the graph concepts of the BGL. Concept maps are instrumental in such cross-domain compositions. The adaptation code involves relatively few lines of code, is transparent to the client, and comes with minimal performance cost.

Note that as part of the adaptation we define a handful of classes that are used as the associated types for the graph concepts. For example, one of the graph concepts requires an associated type `out_edge_iterator`, used by the graph algorithms to traverse the out edges of a node in a graph. The image concepts do not require or guarantee the existence of any types that can directly satisfy the requirements for this associated type, so we create a new small class for this purpose. Objects of this class maintain the state of iteration over a pixel's immediate neighbors. The newly created class does not intrude on the image library, and clients of the image library do not need to explicitly define objects of the iterator class, or even be knowledgeable of its existence. This arrangement serves as an example of the division of labor, in the case when stateful adaptation is needed, between concepts, concept maps, and traditional adaptation via the creation of new classes.

4.1. Background of GIL and BGL

The Generic Image Library is Adobe's open source image processing library, and also part of the *C++ Boost* collection of peer-reviewed C++ libraries (www.boost.org). 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 [56] 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).

Neither the BGL nor the GIL are yet implemented using ConceptC++. We reimplement a subset of these libraries in ConceptC++ for our experiments. We omit support for mechanisms like the BGL's "named parameters" [55, Section 2]. The BGL describes its algorithms' requirements using STL-like concept documentation, and the GIL uses pseudo-code mimicking ConceptC++ to document its concepts; our concepts are translations of these documents into ConceptC++.

For the adaptation layer between the GIL and the BGL we defined concept maps for several GIL concepts, making those concepts models of various graph concepts in the BGL. Using the adaptation layer, many image processing algorithms can be implemented as thin wrappers over the BGL's graph algorithms. We describe the implementation of the adaptation layer, along with the implementations of multiple algorithms. In particular, we focus on the *flood-fill* algorithm. This algorithm modifies the color of a set of contiguous pixels that satisfy a predicate. The implementation of this algorithm performs

```
template <IncidenceGraph G, Buffer Queue, typename Visitor, ColorMap CMap>
  requires BFSVisitor<Visitor, G> && SameType3<G::vertex_t, CMap::key_type, Queue::value_type>
void breadth_first_search (const G& g, const G::vertex_t& s, Queue& Q, Visitor V, CMap Color);
```

Fig. 11. The signature of the **breadth_first_search** function in the BGL. The *same type* constraint guarantees that types of the function arguments are consistent, that is, that the type of the values in the queue argument and the key type of the color map, are the same as the type of the vertices in the graph.

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 partitioning. We also report on using the adaptation layer to image *segmentation*. Graph-based image segmentation refers to a set of techniques for finding a partition of an image by representing the image as a graph, then finding a partition of the graph using, e.g., edge weights or minimal cuts as the partitioning criteria [14,54]. We chose to implement a basic segmentation algorithm based on pixel similarity. The third algorithm we describe is for finding minimal-energy paths between two points in an image. This can be accomplished with the Bellman–Ford [6] shortest path graph algorithm when the input image is represented as a graph.

4.2. GIL–BGL composition

In our adaptation of images to graphs, vertices correspond to pixels and each of the edges connect two vertices corresponding to neighboring pixels. The flood-fill algorithm is essentially a breadth-first graph search. The BGL's breadth-first search algorithm imposes several concept requirements on the types of its inputs, which now have to be satisfied by the image type. We could establish the image-to-graph correspondence directly with concept maps that adapt concrete image types to the BGL graph concepts. However, a broader adaptation for an open ended class of image types is achieved if we adapt generically all types that model GIL image concepts to model BGL concepts. Furthermore, the adaptation is not specific to breadth-first search and 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 Fig. 11. For brevity, in all code examples we omit header includes, namespace prefixes of names from both the GIL and the BGL, and 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 been visited. The **breadth_first_search** function uses four concepts to constrain 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**, which describes the operations of the vertex queue; **BFSVisitor**, which specifies the dictionary of the callback functions; and **ColorMap**, which defines the interface to the data structure storing vertex visitation information.

We focus on the **IncidenceGraph** concept, shown in Fig. 12(a), in the description of the adaptation layer between images and graphs. The **Graph** concept, also in Fig. 12(a), specifies associated vertex and edge types which, via refinement, become requirements of **IncidenceGraph**. Directly, **IncidenceGraph** requires the **out_edges**, **out_degree**, **source**, and **target** operations (lines 10–14). 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 type **out_edge_iterator** on line 7 has its expected meaning.

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 concepts in Fig. 12(b) describe the interface that the GIL imposes on images, and thus provides as images' capabilities. The **ImageView** concept on line 7 provides capabilities of a container. The associated type **value_type** (line 10) is the type of the pixels. The **difference_type** (line 9) represents the offsets between pixel locations, and can also be used as the position of a pixel. The function **dimensions** (line 16) returns the extents of an image. The GIL **ImageView** concept has further requirements, such as an iterator for linear traversal over the sequence of pixels. These capabilities are not necessary for the adaptation to graphs, and are not shown.

The **concept_map** that adapts **ImageView** to **IncidenceGraph** is shown in Fig. 13. Lines 3–5 provide definitions for the associated types of **IncidenceGraph**. We represent vertices as the image's **difference_type**, a point type that specifies the coordinates of a pixel. Edges are pairs consisting of two vertices: the source and target. The **out_edges** function on lines 6–10 constructs a 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, which gives directly the implementation for the **out_degree** function on lines 13–16. The **source** and **target** functions on lines 11 and 12 are trivial.

To arrive at a flood-fill algorithm, we make one additional adaptation: we use a color map tailored for flood-fill, instead of the BGL's default color map for **breadth_first_search**. The color map stores the search state: unseen, in progress, and processed vertices are respectively marked with white, gray, or black. Only white vertices are added to the work queue. The default queue and visitor parameters of **breadth_first_search** that the BGL provides need no customization. With these adaptations, the generic implementation of flood-fill in terms of **breadth_first_search** becomes:

```
concept Graph<typename G> {
2   Regular vertex_t;
    Regular edge_t;
4 };
  concept IncidenceGraph<typename G>
6     : Graph<G> {
    ForwardIterator out_edge_iterator;
8   requires SameType
      <edge_t, out_edge_iterator::value_type>;
10  pair<out_edge_iterator, out_edge_iterator>
    out_edges (const vertex_t&, const G&);
12  size_t out_degree (const vertex_t, const G&);
    vertex_t source (const edge_t&, const G&);
14  vertex_t target (const edge_t&, const G&);
  };
```

a

```
  concept Point<typename P> : Regular<P> {
2   IntegralLike value_type;
    const value_type& operator[](const P&, size_t);
4   value_type& operator[](P&, size_t);
    size_t num_dimensions(P);
6 }
  concept ImageView<typename V> : Regular<V> {
8   Locator locator;
    Point difference_type;
10  Regular value_type;
    requires SameType <value_type, locator::value_type> &&
12    SameType<difference_type, locator::difference_type>;
    const value_type&
14  operator[] (const V&, const difference_type&);
    value_type& operator[] (V&, const difference_type&);
16  difference_type dimensions(const V&);
    size_t size(const V&);
18 };
```

b

Fig. 12. Figure (a) shows the **IncidenceGraph** concept. The **Regular** concept, defined in the draft standard library of ConceptC++, describes a type that is "well-behaved", that is, it can be constructed, destructed, copied, assigned, and compared for equality. Furthermore, these operations adhere to fundamental laws, such as after assigning a value **x** to a variable **y**, then **y == x** returns **true**. The **ForwardIterator** (outlined in Fig. 2) is also a standard concept. Figure (b) shows capabilities provided by the GIL concepts that are relevant for the adaptation. The **IntegralLike** concept (not shown) is a standard concept. The **Locator** concept provides traversal capabilities through the values, i.e., pixels, of GIL images—we omit this concept since **Locator**s offer no generic way to determine the validity of the position of a locator. Our adaptation maintains current location in algorithms with values of the **ImageView**'s **difference_type**.

```
  template <ImageView Img>
2 concept_map IncidenceGraph<Img> {
    typedef Img::difference_type vertex_t;
4   typedef pair<vertex_t, vertex_t> edge_t;
    typedef out_edge_iterator_adapter<vertex_t> out_edge_iterator;
6   inline pair<out_edge_iterator, out_edge_iterator>
    out_edges(const vertex_t& v, const Img& g) {
8     out_edge_iterator first(v, dimensions(g)), last(num_dimensions(g));
      return make_pair(first, last);
10  }
    vertex_t source (const edge_t& e, const Img&) { return e.first; }
12  vertex_t target (const edge_t& e, const Img&) { return e.second; }
    size_t out_degree (const vertex_t& v, const Img&) {
14    pair<out_edge_iterator, out_edge_iterator> iter = out_edges (v);
      return distance(iter.first, iter.second);
16  }
  }
```

Fig. 13. The **concept_map** adapting models of GIL **ImageView** to become models of BGL **IncidenceGraph**.

```
template <ImageView Img, typename P>
  requires Predicate<P, Img::value_type>
void flood_fill(Img& img, const Img::difference_type& seed, P p, const Img::value_type& replacement) {
  if (!p(img[seed])) return;
  vector<Img::difference_type> buffer;

  breadth_first_search(img, seed, buffer, basic_bfs_visitor(), color_map<Img, P>(img, p, replacement));
}
```

The queue and visitor parameters are those used in the BGL by default, but we need to specify them explicitly since our graph library does not implement the BGL's named parameters mechanism that provides support for default values for parameters.

The image segmentation algorithm mentioned in Section 4 can also be implemented in terms of a breadth-first graph search. The task of an image segmentation algorithm is to partition an image into contiguous regions according to some criteria; its central building block is a generic partition algorithm that forms a single partition in an image; repeated invocations with different initial seed locations will segment the entire image. Implementation of the segmentation algorithm uses the same concept map adapters, but a different color map class. The default queue type suffices, but the

BGL's visitor type is customized to respond to the event of discovering a vertex by adding that vertex to a data structure representing a partition.

To further evaluate the usability of cross-domain adaptation between images and graphs, we implemented the *backbone-healing* algorithm [17]. This is a practical image processing algorithm from the TEXTAL automatic model-building protein crystallography package [26], and is used in improving the results of image skeletonization. The algorithm can be implemented in terms of the Bellman–Ford shortest-path graph algorithm, which places stronger requirements on its input graphs than merely **IncidenceGraph**. The BGL's **bellman_ford** algorithm requires the capability to iterate over all edges and all vertices omitting the adjacency structure of the graph, and thus requires that its graph parameter type be a model of the **EdgeAndVertexListGraph** concept.

To allow GIL images to be used as inputs of the **bellman_ford** algorithm, a slightly more capable adaptation is necessary, one that supports iteration over all vertices and all edges of a graph. We define two auxiliary classes, analogous to the **out_edge_iterator_adaptor** class discussed above, that masquerade traversal through an image's pixels and its neighboring pixel pairs as iteration over vertices and edges. With these classes, the adaptation between the GIL's **ImageView** concept and the BGL's **EdgeAndVertexListGraph** concept, is expressed with the concept map:

```
template <ImageView Img> concept_map EdgeAndVertexListGraph<Img> { ... }
```

The implementation of the adaptation is similar to that between the **ImageView** and **IncidenceGraph** concepts; we experienced no notable difficulties in realizing it. We make the full code of the adaptions described above available.[4]

5. Performance

Adaptation mechanisms can have a negative impact on performance. Mitchell et al. [44] give a detailed analysis of a case where multiple inefficient adaptation layers had a major effect on the 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 implemented using concept maps.

We use the flood-fill, segmentation, and backbone-healing algorithms as test cases, and compare the execution times of two different programs for each algorithm. The first program for each algorithm is written directly in terms of the GIL concepts. Essentially, the flood-fill performs a breadth-first search tailored for images, and the segmentation algorithm a series of such searches. The backbone-healing algorithm searches for all shortest paths using the Bellman–Ford algorithm that was directly written for GIL's image types. The second program for each algorithm uses concept maps to adapt GIL concepts to BGL concepts as described in Section 4.2, and uses BGL's **breadth_first_search** function for the flood-fill and segmentation algorithms, and **bellman_ford** for backbone-healing.

We compiled all of the test programs using the ConceptGCC [23] compiler's Alpha 7 Prerelease version[5] with the -O3 flag on two platforms: MacBook Pro (Intel Core 2 Duo), 2.2 GHz, with 2 GB of RAM, and iMac G5 (PowerPC G5), 2.1 GHz, with 1 GB of RAM. The reported timings were obtained by executing the test programs ten times, and computing the average of the measured running times. The test sets for flood-fill and segmentation algorithms consist of 50 square images each, from the size of 20×20 pixels to 1000×1000 pixels. To make the image size directly determine the size of the problem, we use computer-generated images where the number of reachable pixels is proportional to the image size. The test images consist of a maze with vertical, horizontal, and diagonal lines, as well as corners and dead-ends. Fig. 14(a) and (b) show the results for flood-fill, and Fig. 14(c) and (d) the results for segmentation. The test set for the backbone-healing algorithm consists of ten square images, from the size of 10×10 pixels to 100×100 pixels. The test images are a topographical representation of a complex terrain with a non-trivial shortest path between the start and target pixels. The length of the optimal path grows linearly with the side lengths of the image, increasing the size of the problem accordingly; Fig. 14(e) and (f) show the timing results.

For each algorithm we tested, performance of the two implementations is close to the same. The averaged (over different image sizes) *abstraction penalties*, defined as the ratio of an abstracted implementation over a direct implementation [29, Section D.3], due to the adaptation were as follows: for the Intel architecture, flood-fill 0.92, segmentation 0.96, and backbone-healing 1.07; for the PowerPC architecture, flood-fill 1.11, segmentation 1.01, and backbone-healing 1.08. The implementation using cross-domain adaptation thus, in all cases, achieves performance roughly on a par with a hand written GIL algorithm. In two cases the implementation via adaptation was faster. This (close to) zero-overhead adaptation is due to C++'s template compilation model, where specialized code is generated for each different template instantiation. As discussed in Section 2.2, calls to functions defined in concept maps can be statically resolved, and often inlined, allowing the optimizer to see through adaptation layers.

Our tests compared the performance of algorithms written in terms of different data structures: images and graphs. To minimize noise, we were careful to ensure that the compared algorithms nevertheless employed a common strategy, for example for updating work lists. In a few cases, the control structure of the code differs because the direct implementation can take advantage of properties specific to images. Factors such as the use of auxiliary data structures, differences in cache locality, and the success of the compiler's inliner, all have an impact on the final observed performance. The experiments

4 parasol.cs.tamu.edu/groups/pttlgroup/programming-with-c++-concepts.

5 Conceptgcc (GCC) 4.3.0 20070330 (experimental) (Indiana University ConceptGCC Alpha 7 Prerelease), checkout 681.

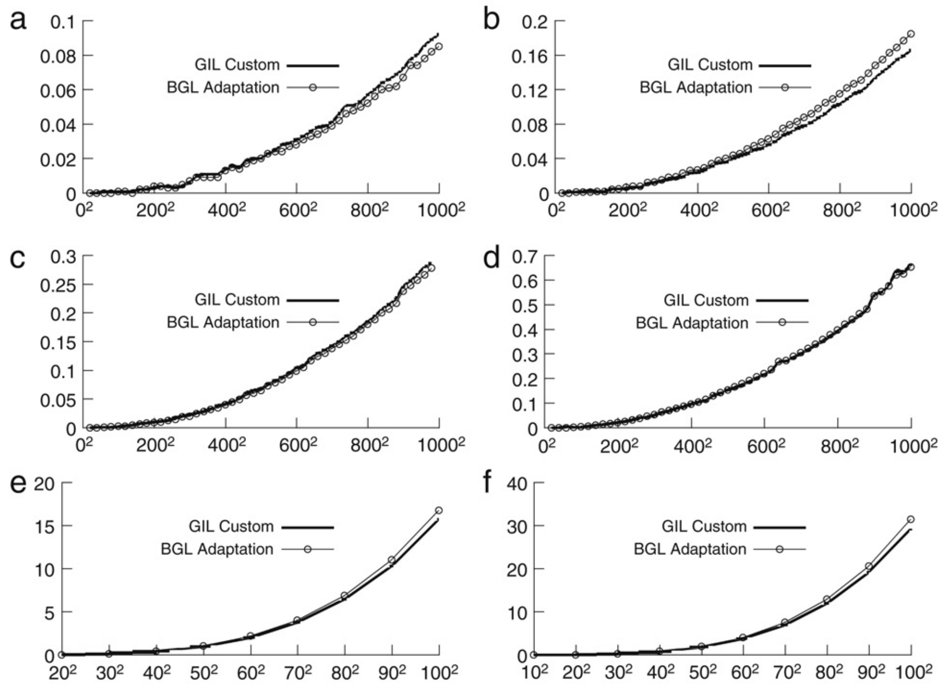

Fig. 14. The timing results for the functions **flood_fill**, (a) and (b), **segmentation**, (c) and (d), and **backbone_healing**, (e) and (f). The results obtained from the Intel architecture are shown in charts (a), (c), and (e), and those from the PowerPC architecture in charts (b), (d), and (f). The *x*-axes of all charts are the number of pixels in the image, the *y*-axes the algorithms' running times in seconds. Each chart depicts the timing results of executing two test programs, the first written directly using GIL concepts (''GIL Custom''), and the second written to use the adaptation between the GIL and the BGL (''BGL Adaptation'').

suggest that the composition mechanism itself incurs no significant penalties; other factors have a larger impact on performance. Furthermore, a generic algorithm in a widely used software library can be expected to be well tuned and tested; reusing such an algorithm even via a complex adaptation layer retains these benefits.

6. Concept maps 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++ 2003.

Concept maps are a non-intrusive mechanism for adapting operations on a type (or collection of types) to model a concept. That is, a given collection of operations and associated types might offer the essential functionality required by an interface (concept), but these operations might not have the required names or signatures, and some associated types might not exist with the correct names. Such a collection can be made conform to the new interface using a concept map. No changes to the original operations or types are needed.

Concept maps adapt types rather than individual objects. They do not offer direct facilities to store state. All state is maintained in the objects whose types concept maps adapt. New types and operations may need to be introduced before the concept map adaptation facilities come into play, if a collection of types does not provide the essential functionality required to model a concept. We saw this in Section 4, where it was necessary to introduce a new class for traversing sets of pixels in an image, in order to satisfy the requirement for an associated type modeling edge iterators. Once all of the necessary functionality has been implemented with appropriate types and functions, concept maps can be applied in their intended role as non-intrusive identity-preserving adapters.

The template system of C++ 2003, 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++ 2003 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, this term covers the ''interface'' language construct found in some languages.) To satisfy the requirements of an interface then means to define 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. Variations of structural subtyping have been proposed as cures for problems of rigid class hierarchies [4,36] but have not found wide use. Outside of mainstream object-oriented languages, Cecil [38] lets one define subtype relationships external to class definition. This feature was found beneficial for adapting existing types for use with generic libraries in a comparative study of programming languages' suitability for generic programming [16]. 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" [34].

The *adapter pattern* [15] 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., [41,61]). Class adapters suffer from *hierarchy hardening* and object adapters from inconsistency problems caused by breaking the state of a single entity into multiple objects [27].

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 [40], 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. If we ignore for the moment concept refinement relationships, some ability to test whether operations are present (via down-casting), and performance optimizations (see Section 3.2), the central design feature of the `poly<>` template that we presented is that it behaves as a type constructor for defining existential types [37], where the hidden type is constrained to be a model of a particular concept. In this regard, `poly<>` is similar to Haskell's "`forall`" construct, which allows the definition of types with a hidden part constrained to be a type belonging to a given type class, or classes.

Concepts are in many ways similar to Haskell type classes [63], 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 [35], 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 [33], a well-established extension, or expressed directly using more recent extensions [9,10].

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. Garcia et al. compare the suitability of different mainstream languages for generic programming [16].

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 qualification) 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. Fig. 15 translates the classic example of accidental conformance [39] to ConceptC++ and to Haskell. The Haskell version is erroneous and is 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, even if there are no conflicting concepts.

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 Fig. 16 illustrates this. The multiplication operator ($*$) for integers is given different semantics in the two concept maps. The first concept map retains the multiplication 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 function's constraints. In our slightly contrived example function, both meanings apply. The fact that concept maps define views that are only active when requested 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.

```
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);
}
```

Fig. 15. Accidental use of the same function name in two different type classes (left column) and in two different concepts (right column).

```
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);
}
```

Fig. 16. 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**.

As an incremental addition to an evolving C++ language, concept maps and concept-based overloading must co-exist, and sometimes compete, with the existing overloading mechanism of C++ 2003. This results in tension between the needs of traditional ad-hoc function template overloading and the desire to treat concrete types and operations as implementation details, and only expose required functionality through concept maps. In Section 7 we illustrate some insidious failure cases that can arise from this competition.

The Scala programming language [47] 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. A fairly faithful emulation of type classes is possible with implicit parameters that represent dictionaries of functions [46]. Furthermore, Scala *views* utilize implicit parameters to non-intrusively define implicit conversions between types, which seems promising for implementing cross-domain compositions like we discussed in Section 4.

C++ 2003 allows the definition of efficient non-intrusive adaptation layers. As an example, we mentioned BGL's transparent adapters for LEDA graphs in Section 2. Breuer et al. [8] 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 [19] to concepts found in the Iterative Eigensolver Template Library [62]. Their implementation is in C++ 2003, and uses overloading and template specialization to achieve the necessary adaptation, not **concept** and **concept_map** constructs of ConceptC++. Non-intrusive adaptation in C++ 2003 relies on a host of tricky template techniques, such as *traits classes* [45] and conditional overloading using the *enable_if* template [31]. Though C++ 2003 can support complex non-intrusive adaptation, the resulting code is brittle; ConceptC++ offers improved support for non-intrusive adaptation.

7. Adaptation and overloading

We have demonstrated many benefits that mechanisms like concepts and concept maps provide when composing software components. There are, however, some stumbling blocks. In this section we illustrate some of the difficulties that arise when using these language features.

One of BGL's graph adapters, **vector_as_graph**, adapts all instances of **vector<list<T>>** to satisfy the requirements in the **IncidenceGraph** table. Adaptation is accomplished by overloading the function templates required by **IncidenceGraph** for **vector<list<T>>**. In ConceptC++, the analogous adaptation is accomplished with a concept map

```
template <class T>
concept_map IncidenceGraph<vector<list<T>>>;
```

C++'s ad-hoc polymorphism defines a partial ordering amongst a set of overloaded functions based on specialization ordering of type patterns. In a nutshell, a function is selected from a set of overloads when it is "at least as specialized as" all other candidates and no other candidate is at least as specialized as it. This relationship considers a type pattern A to be at least as specialized as another type pattern B when A is substitutable for B. ConceptC++ defines an analogous specialization order between concept constraints [30]. For example, one can expect the **size** function below to match and be applied to all **IncidenceGraph**s, calculating the total number of edges in a graph, including calls with arguments of type **vector<list<T>>**, as shown:

```
template <IncidenceGraph G> int size(const G&);
...
vector<list<int>> g;
int total_num_edges = size(g);
```

When executed, the above code will use graph operations to calculate and return the total number of edges in the graph.

For the purposes of determining the partial ordering of constrained function templates, concept constraints, however, are subordinate to type patterns—constraints are only considered in case a "tie-breaker" is needed. For example, the type pattern **vector<T>** is considered strictly more specialized than, say, the constraint **IncidenceGraph<T>**.

Later in the software's life-cycle, another overloaded **size** function may be introduced, for example, to return the size of an arbitrary vector:

```
template <typename A>
int size(const vector<A>& a) { return a.size(); }
```

Now, when we invoke **size** on a **vector<list<T>>** this new overload is considered to be the unique best-matching candidate since the type pattern in the overload defined for **IncidenceGraph<T>** is no more specialized than a plain type variable. That is, the **size(vector<T>)** overload has silently hijacked the call to **size(IncidenceGraph<T>)**. The code will compile but will erroneously only return the number of **lists** in the **vector** representing the graph.

Type patterns are the primary overloading criteria in several other mainstream languages. For example, in C# and Java, overloading is based exclusively on type patterns: constraints on type parameters must be satisfied but they do not affect specialization ordering. Overloading rules in various languages support the notion that the most specific knowledge prevails. ConceptC++ offers two mechanisms for specialization, type patterns and constraints, and by subordinating one mechanism to the other we compromise this notion. A change to the function template partial ordering rules could reinstate the principle that the most specific type knowledge is used for dispatch. In particular, programs with silent failures like the one in the example above could be rejected, if for the purpose of function overload partial ordering, concept map specialization patterns were considered at the same time as function type pattern specializations, rather than as tie-breakers. These issues are currently under consideration in the C++ standards committee.

The unrestricted non-intrusive adaptation allowed by current languages leads to conflicting adaptation layers, and even with the above modification to function overloading rules, surprises and ambiguities seem still possible. A situation akin to the one we demonstrated in C++ arises with Haskell's specialization ordering amongst instance declarations, with language extensions that allow definition of "overlapping instance declarations" [50, Section 3.7].

Non-intrusive adaptation helps to avoid much of the pre-planning and coordination that is necessary in the use of software libraries when component adaptation is intrusive. For this flexibility, one needs not give up efficiency: our work demonstrates that non-intrusive adaptation and efficiency are not mutually exclusive. The possibility of accidentally adapting the same component to a given interface in multiple ways exists, but can be controlled with language designs that support detecting conflicts and offer means for resolving them; further programming language research in this area is required.

8. Conclusions

This paper reports on programming with "concepts", a forthcoming set of new features of C++, and explains their use, benefits, and costs. In particular, concepts offer powerful mechanisms for adapting data types to specific library interfaces—we provide an analysis of this aspect of C++ concepts, and a description of their use in complex cases of non-intrusive library composition. We demonstrate that transparent adaptation of data structures to multiple library interfaces is possible and straightforward. We conclude from our performance evaluations that such adaptations impose minimal penalties.

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 inlineable and optimizeable).

The generic library interfaces defined in ConceptC++ do not directly support run-time polymorphism. We describe the idioms needed to combine run-time polymorphism and concepts. As a result, run-time polymorphism can be introduced

non-intrusively as simply another adaptation layer that can be used by clients of the library if and when they need it. Clients that do not need run-time polymorphism can instantiate library components directly.

The generic programming paradigm, introduced into C++ via the STL, supports the non-intrusive design of efficient families of algorithms specified in terms of common abstractions. There is little language support, however, for rigorously specifying these abstractions. C++ is now evolving to raise such specifications from the level of naming conventions to compiler checkable artifacts. C++0x comes with language support for concepts, and the standard library has been re-specified to take advantage of this, while retaining backward compatibility. The backward compatibility requirement, considered essential for the evolutionary acceptance of C++0x, has had an impact on the concepts in the library, but the effort is nevertheless a significant step forward. The guidance for programming with concepts, as well as the libraries and adaptation idioms presented in this paper, unburdened by legacy compatibility concerns, are further advances towards leveraging the generic programming approach in constructing reusable software libraries. The utility of these idioms will further increase once the concepts language feature becomes widely available: we expect to see algorithm families designed from the ground up with this support in mind, and to see generic programming emerge as a central tool in the design of modern reusable libraries.

Acknowledgments

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. The exchange of experiences with the C++ standards committee has been especially beneficial. This work was supported in part by NSF grant CCF-0541014.

References

[1] Adobe systems, Inc., Adobe Source Libraries, 2008. stlab.adobe.com.
[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, L. Rauchwerger, STAPL: An adaptive, generic parallel C++ library, in: Languages and Compilers for Parallel Computing, in: Lecture Notes in Computer Science, vol. 2624, Springer, 2001, pp. 193–208.
[3] M.H. Austern, Generic programming and the STL: Using and extending the C++ Standard Template Library, in: Professional Computing Series, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
[4] G. Baumgartner, M. Jansche, K. Läufer, Half & half: Multiple dispatch and retroactive abstraction for Java, Technical report OSU-CISRC-5/01-TR08, Ohio State University, 2002.
[5] 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++, April 2006.
[6] R.E. Bellman, On a routing problem, Quart. Appl. Math. 16 (1958) 87–90.
[7] L. Bourdev, H. Jin, Generic Image Library, 2006.opensource.adobe.com/gil.
[8] A. Breuer, P. Gottschling, D. Gregor, A. Lumsdaine, Effecting parallel graph eigensolvers through library composition, in: Workshop on Performance Optimization for High-Level Languages and Libraries, POHLL, in: Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006), Los Alamitos, CA, USA, IEEE Computer Society, April 2006, p. 466. http://doi.ieeecomputersociety.org/10.1109/IPDPS.2006.1639723.
[9] M.M.T. Chakravarty, G. Keller, S. Peyton Jones, S. Marlow, Associated types with class, in: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, NY, USA, 2005, pp. 1–13.
[10] M.M.T. Chakravarty, G. Keller, S. Peyton Jones, Associated type synonyms, in: ICFP '05 : Proceedings of the International Conference on Functional Programming, ACM Press, New York, NY, USA, 2005, pp. 241–253.
[11] C++ Boost, Boost smart pointers library. www.boost.org/libs/smart_ptr.
[12] S. Dickinson, M. Pelillo, R. Zabih, Introduction to the special section on graph algorithms in computer vision, IEEE Trans. Pattern Anal. Mach. Intell. 23 (10) (2001) 1049–1052.
[13] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, S. Schönherr, On the design of CGAL, a computational geometry algorithms library, in: Discrete Algorithm Engineering, Software – Practice and Experience 30 (11) (2000) 1167–1202 (special issue).
[14] P.F. Felzenszwalb, D.P. Huttenlocher, Efficient graph-based image segmentation, Int. J. Comput. Vis. 59 (2) (2004) 167–181.
[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Co., New York, NY, USA, 1995.
[16] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, J. Willcock, An extended comparative study of language support for generic programming, J. Funct. Program. 17 (2007) 145–205.
[17] K. Gopal, R. Pai, T.R. Ioerger, T. Romo, J.C. Sacchettini, TEXTAL: Artificial intelligence techniques for automated protein structure determination, in: Proceedings of the 15th Conference on Innovative Applications in Artificial Intelligence, IAAI, 2003, pp. 93–100.
[18] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G.D. Reis, A. Lumsdaine, Concepts: Linguistic support for generic programming in C++, in: OOPSLA '06: Proceedings of the 2006 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, 2006, pp. 291–310.
[19] D. Gregor, A. Lumsdaine, Lifting sequential graph algorithms for distributed-memory parallel computation, in: OOPSLA '05: Proceedings of the 2005 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2005, pp. 423–437.
[20] D. Gregor, M. Marcus, T. Witt, A. Lumsdaine, Foundational concepts for the C++0x standard library, Technical Report N2677 = 08-0187, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming language C++, 2008.
[21] D. Gregor, J. Siek, Implementing concepts, Technical Report N 1848=05-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming language C++, August 2005. www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1848.pdf.
[22] D. Gregor, B. Stroustrup, J. Widman, J. Siek, Proposed wording for concepts (revision 8), Technical Report N2741=08-025, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming language C++, August 2008.
[23] D. Gregor, ConceptGCC: Concept extensions for C++, 2005. http://www.generic-programming.org/software/ConceptGCC.
[24] H.E. Hinnant, D. Abrahams, P. Dimov, A proposal to add an rvalue reference to the C++ language, Technical Report N 1690=04-0130, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming language C++, September 2004.www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1690.html.
[25] H.E. Hinnant, P. Dimov, D. Abrahams, A proposal to add move semantics support to the C++ language, Technical Report N1377=02-0035, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming language C++, September 2002. www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm.

[26] T.R. Holton, T.R. Ioerger, J.A. Christopher, J.C. Sacchettini, TEXTAL: A pattern recognition system for interpreting electron density maps, in: Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology, ISMB, 1999, pp. 130–137.
[27] IBM Research, Subject-oriented programming and the adapter pattern. www.research.ibm.com/sop/sopcadap.htm.
[28] International organization for standardization, ISO/IEC 14882:2003: Programming languages: C++, 2nd ed., Geneva, Switzerland, 2003, p. 757.
[29] Technical report on C++ performance, Technical Report N1487=03-0070, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming language C++, 2003.
[30] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, Algorithm specialization in generic programming: Challenges of constrained generics in C++, in: PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 2006, pp. 272–282.
[31] J. Järvi, J. Willcock, A. Lumsdaine, Concept-controlled polymorphism, in: F. Pfennig, Y. Smaragdakis (Eds.), GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering, in: LNCS, vol. 2830, Springer Verlag, 2003, pp. 228–244.
[32] M. Jazayeri, R. Loos, D. Musser, A. Stepanov, Generic programming, Report of the Dagstuhl seminar on generic programming, Schloss Dagstuhl, Germany, 1998. www.dagstuhl.de/Reports/98171.pdf.
[33] M.P. Jones, Type classes with functional dependencies, in: ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems, in: Lecture Notes in Computer Science, vol. 1782, Springer-Verlag, New York, NY, 2000, pp. 230–244.
[34] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An overview of AspectJ, in: J.L. Knudsen (Ed.), Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP'01, in: Lecture Notes in Computer Science, vol. 2072, Springer-Verlag, London, UK, 2001, pp. 327–353.
[35] R. Lämmel, K. Ostermann, Software extension and integration with type classes, in: GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, ACM Press, New York, NY, USA, 2006, pp. 161–170.
[36] K. Läufer, G. Baumgartner, V.F. Russo, Safe structural conformance for Java, Comput. J. 43 (6) (2000) 469–481.
[37] K. Läufer, M. Odersky, Polymorphic type inference and abstract data types, ACM Trans. Program. Lang. Syst. 16 (5) (1994) 1411–1430. http://doi.acm.org/10.1145/186025.186031.
[38] 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, ACM Press, New York, NY, USA, 1998, pp. 388–411.
[39] B. Magnusson, Code reuse considered harmful, J. Object-Oriented Program. 4 (3) (1991) 8.
[40] M. Marcus, J. Järvi, S. Parent, Runtime polymorphic generic programming—Mixing objects and concepts in ConceptC++, in: K. Davis, J. Striegnitz (Eds.), Multiparadigm Programming 2007: Proceedings of the MPOOL Workshop at ECOOP '07, 2007. homepages.fh-regensburg.de/mpool/.
[41] M. Mattsson, J. Bosch, M.E. Fayad, Framework integration problems, causes, solutions, Commun. ACM 42 (10) (1999) 80–87.
[42] B. McNamara, Y. Smaragdakis, Static interfaces in C++, in: First Workshop on C++ Template Programming, Erfurt, Germany, 2000. oonumerics.org/tmpw00/.
[43] K. Mehlhorn, S. Näher, The LEDA Platform of Combinatorial and Geometric Computing, Cambridge University Press, 1999.
[44] N. Mitchell, G. Sevitsky, 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; As technical report 06–12 of Rensselaer Polytechnic Institute, Computer Science Department, October 2005.
[45] N.C. Myers, Traits: A new and useful template technique, C++ Report, 7(5), 1995, pp. 32–35.
[46] M. Odersky, Poor man's type classes, Presentation at the meeting of IFIP WG 2.8, Functional Programming, July 2006. lamp.epfl.ch/~odersky/talks/wg2.8-boston06.pdf.
[47] M. Odersky, The Scala Language Specification: Version 2.0, Draft March 17, 2006. http://scala.epfl.ch/docu/files/ScalaReference.pdf.
[48] S. Parent, Beyond objects: Understanding the software we write, Presentation at C++ Connections, 2005. opensource.adobe.com/wiki/index.php/Image:Regular_object_presentation.pdf.
[49] 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.
[50] S. Peyton Jones, M. Jones, E. Meijer, Type classes: An exploration of the design space, in: Proceedings of the Second Haskell Workshop, 1997. citeseer.ist.psu.edu/peytonjones97type.html.
[51] P. Pirkelbauer, S. Parent, M. Marcus, B. Stroustrup, Runtime concepts for the C++ standard template library, in: SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing, ACM, New York, NY, USA, 2008, pp. 171–177.
[52] W.R. Pitt, M.A. Williams, M. Steven, B. Sweeney, A.J. Bleasby, D.S. Moss, The bioinformatics template library–generic components for biocomputing, Bioinformatics 17 (8) (2001) 729–737.
[53] U. Shani, Filling regions in binary raster images: A graph-theoretic approach, SIGGRAPH Comput. Graph. 14 (3) (1980) 321–327.
[54] J. Shi, J. Malik, Normalized cuts and image segmentation, IEEE Trans. Pattern Anal. Mach. Intell. 22 (8) (2000) 888–905.
[55] J. Siek, L.-Q. Lee, A. Lumsdaine, The Boost Graph Library: User Guide and Reference Manual, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
[56] J. Siek, A. Lumsdaine, L.-Q. Lee, Boost Graph Library, Boost, 2001. www.boost.org/libs/graph.
[57] J. Siek, A. Lumsdaine, The matrix template library: Generic components for high-performance scientific computing, Comput. Sci. Eng. 1 (6) (1999) 70–78.
[58] J. Siek, A. Lumsdaine, Concept checking: Binding parametric polymorphism in C++, in: First Workshop on C++ Template Programming, October 2000. oonumerics.org/tmpw00/.
[59] Silicon Graphics, Inc., SGI implementation of the Standard Template Library, 2004. http://www.sgi.com/tech/stl/.
[60] A. Stepanov, M. Lee, The Standard Template Library, Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, 1994. www.hpl.hp.com/techreports.
[61] C. Szyperski, Component software: Beyond Object-Oriented Programming, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.
[62] M. Troyer, P. Dayal, R. Villiger, The iterative eigensolver template library. www.comp-phys.org/software/ietl/.
[63] P. Wadler, S. Blott, How to make ad-hoc polymorphism less ad-hoc, in: ACM Symposium on Principles of Programming Languages, ACM, 1989, pp. 60–76.